uc3m | Universidad Carlos III de Madrid

Grado en Ingeniería de Sistemas de Comunicaciones

Curso Académico 2018-2019

*Trabajo Fin de Grado*

# Análisis de Rendimiento de Autoencoders en Entornos Big Data

---

Álvaro de los Reyes Guío

Tutor

Harold Yesid Molina Bulla

8 de julio de 2019

# RESUMEN

La revolución tecnológica en la que la sociedad se haya inmersa en los últimos años ha supuesto el surgimiento de nuevos avances, como la ciencia de datos o el aprendizaje profundo que ya existían, pero cuyo potencial no había sido descubierto. Esos campos fueron revitalizados tan pronto como la tecnología disponible fue capaz de cumplir sus necesidades en cuestiones de capacidad computacional y velocidad.

El interés en estas especialidades se ha incrementado también con la emergencia de las últimas técnicas y conceptos, como el *Big Data* y la quinta generación de redes móviles (5G), que ven cómo sus propias áreas de aplicación se ven fuertemente influenciadas por el aprendizaje máquina y las redes neuronales; todo ello en una sociedad en la que cada dispositivo estará conectado con el resto, transmitiendo una gran cantidad de información, y dando forma a un mundo interconectado.

No obstante, unidas a la importancia de estas líneas de trabajo vienen numerosas herramientas con las que diseñar e implementar arquitecturas en una amplia variedad de lenguajes y librerías. Es el caso, por ejemplo, de *TensorFlow*, *Theano*, *Keras* y *Spark*, que serán de enorme relevancia a lo largo de este documento, ya que el propósito de éste consiste en tratar con estos recursos, para analizar su rendimiento al ser aplicados a un tipo concreto de red neuronal, el *autoencoder*, cuyo funcionamiento radica en aprender la representación (las características principales) de una entrada, con el objetivo de regenerar esos datos originales a partir de esos rasgos adquiridos.

**Palabras clave:** *Autoencoders*; redes neuronales; aprendizaje máquina; aprendizaje profundo; extracción de características; reducción dimensional; aprendizaje; análisis de rendimiento; TensorFlow; Theano; Keras; Spark; Elephas.

# DEDICATORIA

En el transcurso de estos meses de esfuerzo y trabajo constante he aprendido muchas cosas, tanto a nivel técnico como personal, que espero queden reflejadas en este trabajo y me acompañen a lo largo de mi vida. Durante todo este tiempo he recibido el apoyo y la ayuda de un buen número de personas, sin las cuales no podría haber llegado hasta aquí.

En primer lugar quiero agradecer a mi tutor, Harold, toda su dedicación, esfuerzo y paciencia en los momentos más complicados, en los que siempre estuvo disponible cuando le necesitaba. Además, debo reconocerle su capacidad para transmitirme algunos de los conceptos más complejos que se me han presentado durante la elaboración del trabajo. Quiero hacer mención también al resto de profesores que me han impartido clase en estos años, gracias a los cuales he adquirido muchos conocimientos, sin los cuales no me habría sido posible llevar a cabo este proyecto, y menos aún terminar el grado de Sistemas de Comunicaciones.

En segundo lugar deseo dar las gracias a mis compañeros de clase, con los que tanto he compartido en este tiempo, por su amistad, generosidad y simpatía.

Quiero agradecer, por último, a mi familia, y especialmente a mis padres, su apoyo constante y el hecho de que me hayan dado la oportunidad de estudiar lo que he querido.

Sobra decir que el éxito y los frutos de este trabajo son también vuestros.

Muchas gracias a todos,

Álvaro

# ÍNDICE GENERAL

# ÍNDICE DE FIGURAS

# ÍNDICE DE TABLAS

# CONVENCIONES Y SIGLAS

A continuación se comenta una serie de decisiones que se han tomado sobre el estilo a utilizar en el trabajo.

En cuanto a las palabras en otros idiomas, se empleará el tipo de letra cursiva, como en *batch*, y se intentará emplear en la medida de lo posible la terminología española para denominar algunos aspectos técnicos, exceptuando los casos en que esta no se emplee de forma generalizada en la literatura científica. Es el caso, por ejemplo, de *Big Data* o de *autoencoder*.

Para los números, se utilizará el punto (.) para indicar la posición decimal, mientras que la coma (,) se usará puntualmente para separar los millares.

Por último, el código Python que se inserta en varios puntos a lo largo del trabajo tendrá el siguiente estilo:

```
1
2  print("Codigo de prueba")
```

CAPÍTULO 1

# INTRODUCCIÓN

## 1.1. Motivación

La revolución tecnológica en la que se haya inmersa la sociedad en los últimos años ha alcanzado un nuevo hito con la explosión del *data science* o ciencia de datos, que se halla en la intersección entre la ingeniería, la informática y la estadística, y que comprende campos emergentes como el aprendizaje automático (*machine learning*) o la analítica predictiva, que se han visto impulsados y revitalizados por el avance técnico [1].

En el último lustro, el auge de este *data science* se ha visto requerido por el desarrollo tanto de la tecnología en general, como de las comunicaciones móviles (red 5G); hacia una situación de conectividad total (internet de las cosas o *Internet of Things*, *IoT*), ya no sólo de dispositivos tradicionalmente conectados entre sí (teléfonos móviles, ordenadores, tabletas...), sino de todo un repertorio de aparatos electrónicos como pueden ser electrodomésticos, sistemas de calefacción y seguridad en el ámbito doméstico; o de iluminación, gestión de residuos y tráfico en el urbano [2].

Esa evolución hacia un mundo ampliamente conectado lleva consigo un aumento exponencial en el volumen de datos que se pueden manejar (*Big Data* [3]), debido a la gran cantidad de información que sensores y antenas dentro de cada aparato o sistema previamente mencionado es capaz de proporcionar. Sectores como la salud, el sector público, el transporte, el entretenimiento o la seguridad se ven claramente favorecidos ya en la actualidad, y sus beneficios en el futuro serán mucho mayores, estando las próximas tendencias ligadas a ámbitos como el control de voz, la protección de la información o la inteligencia

artificial conjuntamente con herramientas de análisis de datos [4].

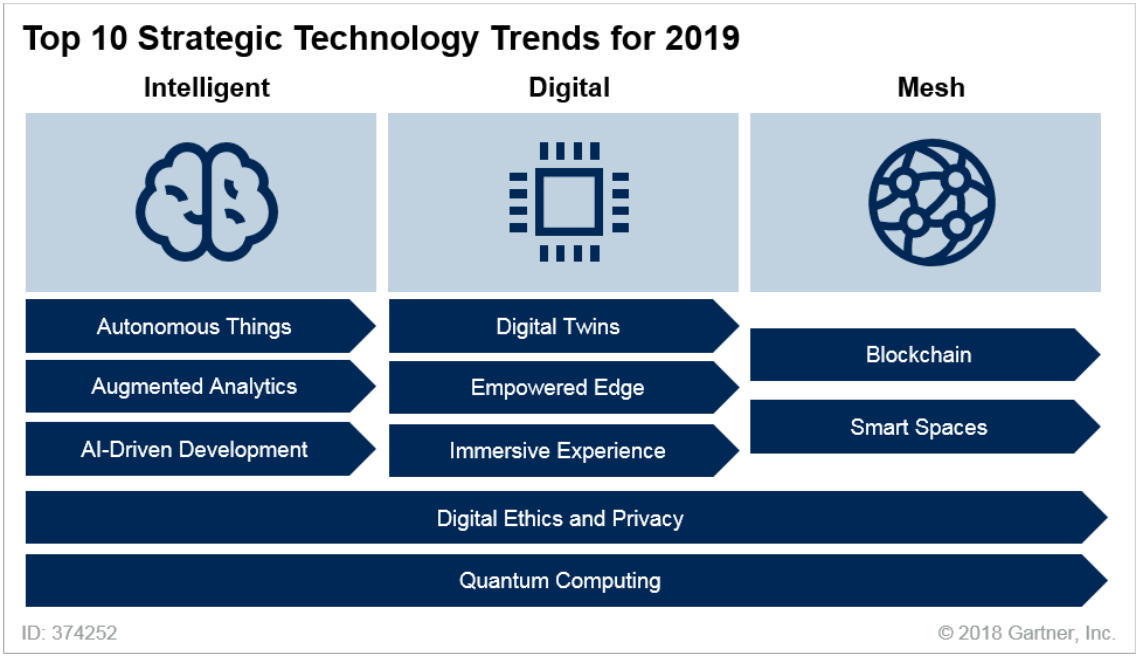



Fig. 1.1. Las 10 tendencias tecnológicas principales para 2019 [5]

Es precisamente en la metodología de análisis donde se ha dado un gran salto evolutivo en esta década, propiciado por la abundancia de datos y la necesidad de operar con ellos para extraer información. Este desarrollo ha afectado especialmente a herramientas como las redes neuronales (*neural networks*), que comenzaron a utilizarse a mediados del siglo pasado [6], cuya relevancia aumentó un tiempo después gracias a métodos de entrenamiento como la propagación hacia atrás (*backpropagation*) [7]; y cuya explosión definitiva vino en la década de los 2010 de la mano de importantes avances en software y hardware, como las unidades de procesamiento gráfico o GPUs (*Graphics Processing Unit*) [8], que permitieron implementar algoritmos más complejos.

## 1.2. Objetivo del trabajo y metodología

El propósito de este documento es realizar un análisis comparativo entre varias de las herramientas empleadas actualmente para la programación de redes neuronales, con el fin de estudiar su rendimiento en un modelo concreto: los *autoencoders* o autocodificadores. Para llevar a cabo ese objetivo se evaluará el desempeño del mismo caso en las siguientes condiciones, todas ellas programadas sobre el lenguaje Python [9]:

- Librería *TensorFlow* [10].

- Librería *Theano* [11].
- Agregador *Keras* [12] sobre *TensorFlow* o *Theano*.
- *Keras* sobre *TensorFlow* o *Theano* mediante *Elephas* [13] en *Spark* [14] (*PySpark* [15]).

En la simulación se utilizará el conjunto de datos (*dataset*) *MNIST* (*Modified National Institute of Standards and Technology*) [16], así como *Fashion-MNIST* [17] al tratarse de una colección de imágenes algo más compleja que la anterior, por lo que el estudio será más completo. Concretamente, *Fashion-MNIST* es una base de datos de imágenes de ropa, y *MNIST* de dígitos escritos a mano, que supone el repertorio de imágenes más empleado como punto de partida al introducirse al *data science*, a pesar de ser algo antiguo [18]. A continuación aparecen dos muestras de ambos conjuntos:

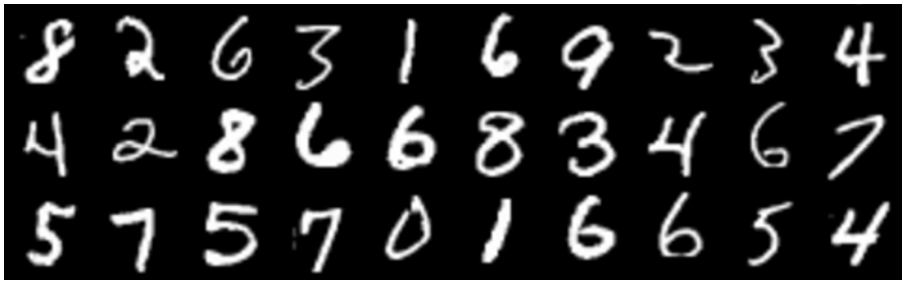


Fig. 1.2. Muestra del banco de imágenes de *MNIST* [16]

Fig. 1.3. Muestra del banco de imágenes de *Fashion-MNIST* [17]

Una vez obtenidos los resultados del entrenamiento, se evaluará además el tiempo consumido, para realizar un estudio comparativo completo entre todas las metodologías empleadas, y seleccionar así la que mejor rendimiento en cuanto a error y a tiempo ofrezca para el caso que se está tratando.

## 1.3. Estructura de la memoria

La memoria del trabajo que se presenta está estructurada de acuerdo al formato *IMRyD*: *Introducción*, *Métodos*, *Resultados y Discusión*, frecuente en artículos de investigación científica [19], constando concretamente de los siguientes apartados:

- Introducción: pone en contexto el tema del trabajo en cuanto a la motivación y los objetivos del mismo. Se incluye además una guía con las convenciones en cuanto a estilo que se van a seguir.
- Estado del arte: describe la situación de la tecnología relacionada con el presente trabajo en la actualidad, indicando las herramientas que se encuentran disponibles; así como un prólogo de carácter histórico a modo de ayuda para entender mejor cómo se han alcanzado esas circunstancias.
- Análisis del problema: de la sección precedente se deriva una cuestión a la cual este documento pretende ofrecer resolución, cuyo estudio previo se incluye en este apartado, al igual que la extensión que pretende abarcar.
- Diseño e implementación de la solución: en concordancia con el capítulo anterior, aquí se detallan las decisiones tomadas en cuanto a qué valores, algoritmos, metodologías... se utilizarán en la solución.
- Experimentos y análisis de resultados: comprende y ahonda en el objetivo primero de esta memoria, ofrecer una comparativa entre multitud de casos para las diferentes herramientas empleadas en la programación de redes neuronales, concretamente *autoencoders*.
- Conclusiones: sirve como respuesta a la cuestión planteada en el estudio y recogida en los objetivos, considerando y evaluando si lo que se planteó como propósito ha sido cubierto.
- Anexos: en este apartado se encuentran un comentario detallado sobre el marco regulador que rodea la tecnología planteada en el trabajo, el entorno socioeconómico en el que se encuadra la misma, con la planificación y el presupuesto que se desprenden del documento; y un resumen en inglés de los contenidos del proyecto.

CAPÍTULO 2

# ESTADO DEL ARTE

## 2.1. Introducción a las redes neuronales

### 2.1.1. Contexto histórico

Como se mencionó en la introducción, los primeros artículos que sentaban las bases de lo que actualmente se conoce como *data science*, se publicaron en la década de 1940. Concretamente, Warren S. McCulloch y Walter Pitts publicaron en 1943 un artículo titulado *A logical calculus of the ideas immanent in nervous activity* [6], donde se planteaba un modelo neuronal que realizaba una operación de sumatorio sobre entradas binarias, y retornaba un valor de 1 en caso de que la suma excediera un cierto umbral, o un 0 en caso contrario (función de activación). Esta idea fue tomada por Frank Rosenblatt [20] en 1958, quien desarrolló el algoritmo conocido como *perceptrón* [21], añadiéndole un mecanismo de aprendizaje, basado en el trabajo de Donald Hebb [22], descrito de la siguiente forma:

> *Let us assume then that the persistence or repetition of a reverberatory activity (or "trace") tends to induce lasting cellular changes that add to its stability. (...) When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.*
>
> Regla de Hebb, de *The organization of behavior. A neuropsychological theory* [22]

Esta regla destaca la fuerte relación de asociación que se formará, y consecuentemente fortalecerá, entre dos neuronas o células que tienden a activarse juntas o, en el caso contrario, la inactivación de algunas de esas relaciones cuando no formen parte del patrón.

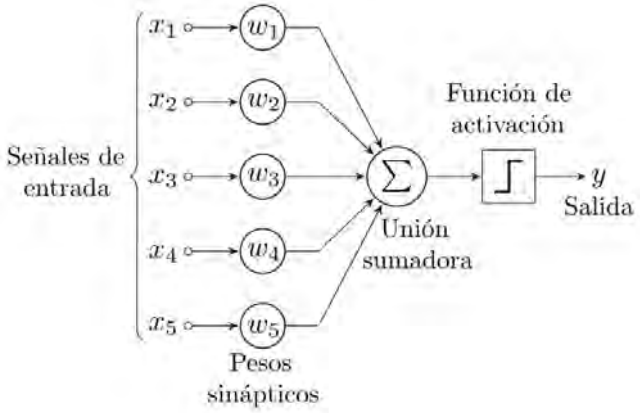



Fig. 2.1. Diagrama de un perceptrón con cinco señales de entrada [23].

La idea desarrollada por Rosenblatt fue implementada, primero en software en una computadora IBM 704 [24], y posteriormente en hardware como la *Mark I Perceptron*, que disponía de un conjunto de 20x20 fotocélulas que hacía las veces de entrada a la cual se añadían unos pesos, regulados por potenciómetros, y ajustados por motores eléctricos para la parte del aprendizaje [24]. De esta forma, agrupando múltiples perceptrones en una capa, recibiendo todos la misma entrada y retornando cada uno de ellos una salida distinta, es como se pueden realizar tareas de clasificación; además, la unión de varias capas origina el concepto de red neuronal artificial (o *Artificial Neural Network*, ANN) [25].

Sobre el trabajo de Rosenblatt, Bernard Widrow planteó la supresión de la función de activación en el perceptrón, de tal forma que el aprendizaje, el hallazgo de los pesos idóneos, se realizaba de forma continua estudiando la variación del error mediante el algoritmo del gradiente descendente o LMS (*Least-Mean-Square*) [26], según estos oscilaban. A este modelo Widrow le llamó *ADALINE* (*ADAptative LINear Element*) [27], y sus principales diferencias con el perceptrón vienen descritas gráficamente en la siguiente imagen:

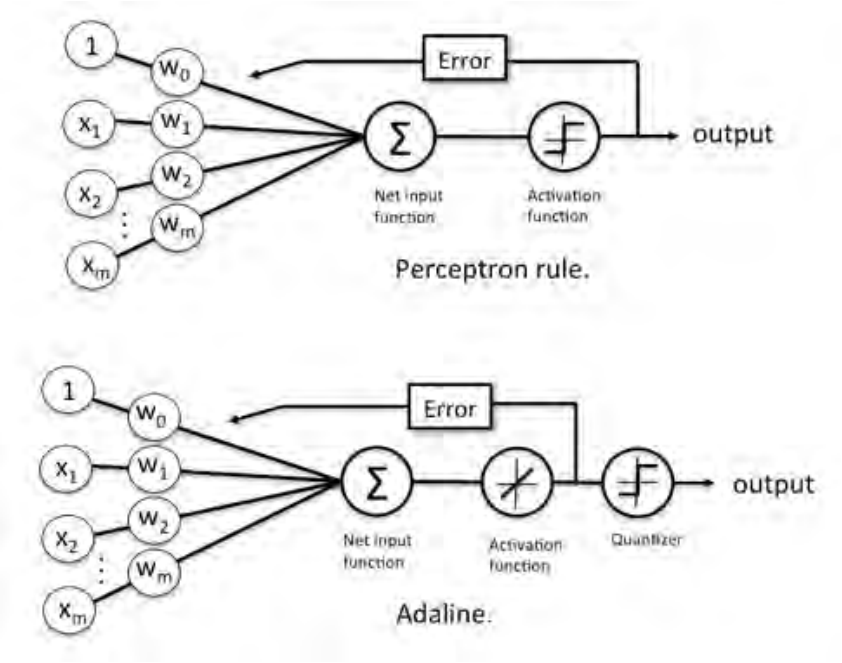



Fig. 2.2. Comparativa entre los modelos de red neuronal perceptrón y ADALINE [28] *(Nota del autor: "Note that I inserted the "activation function" in Adaline just for illustrative purposes; here, this activation function is simply the identity function")*

No obstante, todos los avances realizados hasta el momento se toparon con una serie de artículos, que mostraban cierto escepticismo ante las limitaciones de los perceptrones [29], y concluían que ese acercamiento a la inteligencia artificial llevaba inequívocamente a un punto muerto. La repercusión de estos trabajos provocó una reducción durante la década de 1970 en la financiación y, en consecuencia, en las publicaciones, lo cual se conoce como el *primer invierno de la IA* [30].

La situación originada por el invierno de la IA dificultó, aunque no impidió completamente, que se siguiera investigando; fue de hecho uno de los avances realizados en esta época lo que marcó el fin de la misma. El uso en redes neuronales de la propagación hacia atrás, o *backpropagation*, fue propuesto por primera vez por Paul Werbos en 1974 [31], pero publicado en 1982 debido al poco interés que existía ya en la materia. Dicha técnica suponía una generalización del gradiente descendente a las redes con múltiples capas, actualizando los errores *hacia atrás*, desde la capa de salida hacia la de entrada [26]. En 1986 este trabajo es rescatado por David Rumelhart, Geoffrey Hinton y Ronald Williams, que en su artículo *Learning representations by back-propagating errors* [32] profundizaron en este algoritmo, al tiempo que replicaban los argumentos esgrimidos una década atrás contra los perceptrones.

Este nuevo hito en la historia de las redes neuronales reavivó la esperanza en su futuro,

de tal forma que sólo 3 años después, en 1989, Yann LeCun [33] llevó a cabo la primera demostración de aplicaciones de las redes neuronales en los AT&T Bell Labs: *Backpropagation Applied to Handwritten Zip Code Recognition* [34]. En este modelo la primera capa oculta, la inmediatamente posterior a la de entrada, es convolucional, caracterizada por reconocer en una imagen sus rasgos más significativos, como pueden ser los bordes; y mediante la cual se consigue escalar mejor en número de conexiones y conseguir así una mayor eficiencia. Las otras dos capas ocultas son, sin embargo, redes neuronales normales que utilizan esas características principales para determinar de qué número se trata:

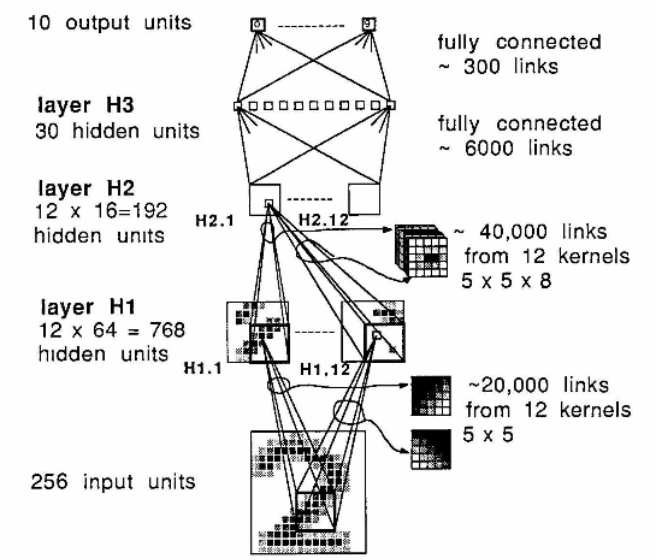



Fig. 2.3. Ejemplo de funcionamiento de la red neuronal descrita por LeCun [34]

El siguiente ámbito al que se aplicarían las redes neuronales sería al de la compresión, es decir, una representación con un menor uso de datos desde la que se pueda reconstruir el archivo original [35]. Esta red se conoce como *autoencoder* y basa su funcionamiento en replicar a la salida la entrada que se ha insertado. Introduciendo entre ambos extremos una capa oculta con menos nodos se consigue que desde el inicio hasta ese punto exista una codificación, mientras que desde entonces hasta el final los datos se decodifican, obteniendo la entrada [36] [37]:

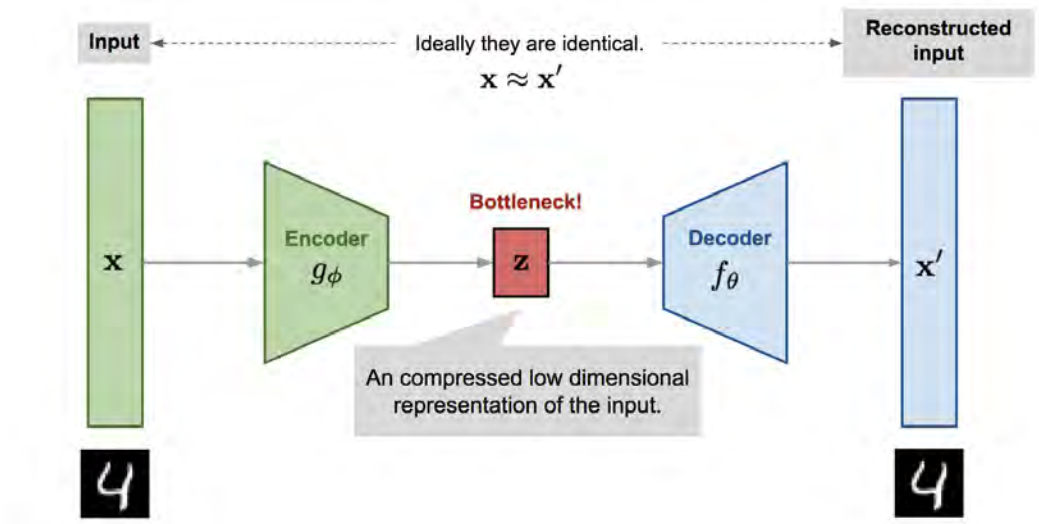



Fig. 2.4. Arquitectura de un autoencoder [36]

Además de los autoencoders, durante los sucesivos años se publicaron nuevos trabajos que profundizaban en el uso de las redes neuronales en otros campos. Al margen de algoritmos como la *máquina de Boltzmann* [38] o las *redes bayesianas* [39], publicaciones como *ALVINN: an Autonomous Land Vehicle in a Neural Network* [40], donde se trataba el aprendizaje de un sistema para controlar un coche; o *Reinforcement Learning for Robots Using Neural Networks*, en la que se discutía la enseñanza a robots de acciones como atravesar puertas en un tiempo mucho menor al que se necesitaba entonces; ponían de manifiesto que el sector se encontraba en un gran estado de forma [37].

No obstante, uno de los mayores avances realizados en el ámbito de la inteligencia artificial sería el que terminaría conduciendo a un nuevo *invierno de la IA*. Gerald Tesauro desarrolló en 1995 una red neuronal aplicada al juego del backgammon, a la que llamó *TD-Gammon* [41], y que logró convertirse en uno de los mejores jugadores a nivel mundial, demostrando que era posible superar a los seres humanos en tareas de cierta complejidad. Sin embargo, al abordar otros juegos como el ajedrez o el go, quedó de manifiesto la deficiente capacidad del hardware en relación a la tarea que querían acometer, al tener que "pensar" varios turnos por delante [42].

Durante este nuevo período, que transcurrió desde mediados de los 90 hasta mitad de la década de los 2000, prosperaron nuevos conceptos como las redes neuronales recurrentes (*RNN*: *Recurrent Neural Nets*), que se aplicaron al reconocimiento de voz [43]; las unidades LSTM (*Long Short-Term Memory*), empleadas dentro de las anteriores; las *máquinas de vectores de soporte*, o *SVM*: *Support Vector Machines*; o los *bosques aleatorios* (*random forests*); muchos de los cuales continúan empleándose en la actualidad.

Por otra parte, las restricciones a la financiación de las redes neuronales hizo que los investigadores renombrar sus nuevos trabajos bajo la denominación de *aprendizaje profundo*, o *deep learning* [44] [45] [46]; siendo además uno de estos, de 2006, el que renovó el interés por la materia, *A fast learning algorithm for deep belief nets* [45], proponiendo una alternativa más eficiente a la inicialización aleatoria de los pesos dentro de una red con muchas capas. Otras publicaciones, como *Scaling learning algorithms towards AI* [47], o *Deep Belief Networks for phone recognition* [48], también ayudaron a relanzar una corriente que llega hasta la actualidad.

Todas estas investigaciones no habrían sido capaces de poner fin al *segundo invierno de la IA* de no haber contado con un gran aumento de la potencia computacional, no sólo con la mejora de la capacidad de las CPU, sino con la paralelización de las mismas, y en mayor medida, con la introducción de las GPU (*Graphics Processing Unit*, unidad de procesamiento gráfico) [49], mediante las cuales consiguieron un incremento de velocidad de hasta 70 veces con respecto a una CPU [50]. Esta evolución tecnológica propició también la eclosión de un nuevo concepto, el *Big Data*, que hace referencia a la existencia de una gran cantidad de datos por explotar y analizar, que está presente fundamentalmente en las grandes empresas tecnológicas, como Microsoft, Google o IBM; como se pone de manifiesto en el artículo *Deep Neural Networks for Acoustic Modeling in Speech Recognition* [51], de 2012, en el que colaboraron todas las anteriores con la Universidad de Toronto, pionera y líder en el estudio de las redes neuronales.

El éxito actual del aprendizaje profundo viene dado por la incapacidad de otros métodos de lograr tasas de error tan bajas como las que se consiguen mediante redes neuronales, para tareas como la clasificación de imágenes o el reconocimiento de objetos [44]. Además, el impulso que se lleva a cabo desde las grandes empresas tecnológicas hace presagiar un futuro en el que las redes neuronales estarán muy presentes en el día a día [52].

### 2.1.2. Modelos de redes neuronales

A continuación se presentarán varios de los principales tipos de redes neuronales que existen. Las imágenes que acompañan a las explicaciones tienen el mismo origen ([53]), donde se puede encontrar además la siguiente leyenda que explica los códigos de colores y símbolos que tiene cada nodo:

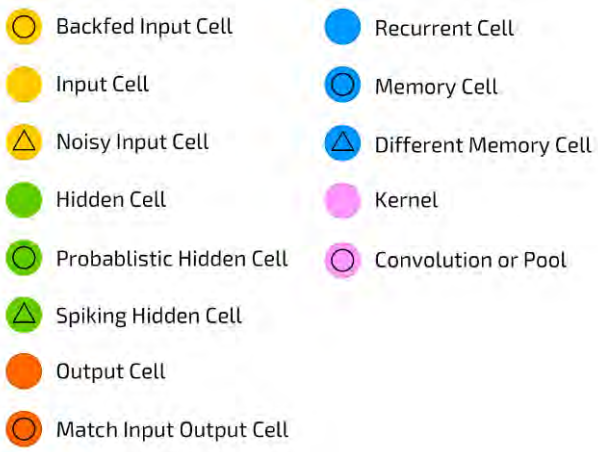



Fig. 2.5. Leyenda para los modelos de redes neuronales [53].

- Perceptrón: es el más simple y también el más antiguo. Consiste en recibir unas entradas, juntarlas, aplicar una función de activación y enviarlas a la salida directamente o a través de varias capas (perceptrón multicapa), que generalmente constará únicamente de un nodo [46].

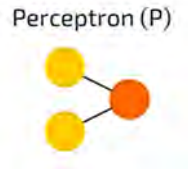



Fig. 2.6. Modelo de perceptrón [53]

- Prealimentada (*Feed Forward*): es una de las más antiguas y consiste en una conexión completa entre las capas, que generalmente son tres: entrada, una oculta y salida. Además, no presenta bucles y suele ser entrenada mediante propagación hacia atrás (*backpropagation*) [53].


Fig. 2.7. Modelo de red prealimentada (*Feed Forward Neural Network*) [53]

- Recurrente: la idea principal es que sus capas ocultas están formadas por nodos recurrentes, que reciben su propia salida con un retraso determinado. Se utilizan para procesar datos secuenciales, de tal forma que, por ejemplo, el significado de una palabra puede venir dado por el contexto [46].

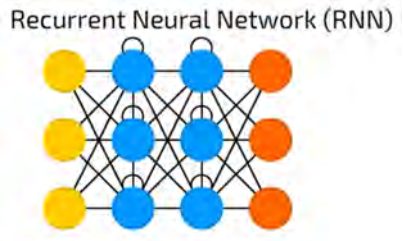



Fig. 2.8. Modelo de red recurrente [53]

- *Long Short-Term Memory*: utilizan nodos de memoria, que pueden almacenar una pequeña cantidad de información anterior, lo cual hace que sean empleadas actualmente para aplicaciones de reconocimiento de voz, como por ejemplo *Alexa*, de *Amazon* [54] [46].

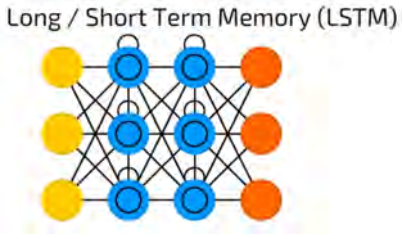



Fig. 2.9. Modelo de red *Long Short-Term Memory* [53]

- Convolucionales Profundas: están formadas por filtros (*kernel*), que procesan los datos de entrada, y nodos convolucionales, que los van comprimiendo en capas sucesivas, reduciendo así características innecesarias [46]. Se suelen emplear para reconocimiento de imágenes, operando en pequeñas ventanas, que se van desplazando por ellas.

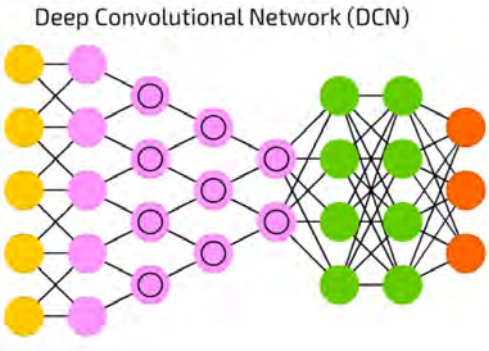



Fig. 2.10. Modelo de red convolucional profunda [53]

- *Autoencoder*: está entrenada para intentar copiar la entrada a la salida, de tal forma que consta de dos partes, codificador y decodificador [46]. Una explicación más

extensa se puede encontrar en la sección 2.2.

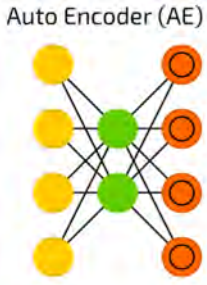



Fig. 2.11. Modelo general de *autoencoder* [53]

- Cadena de Markov: es un concepto de grafo en el que existen varios estados, con una probabilidad de pasar a otro distinto o de permanecer en el mismo [26].

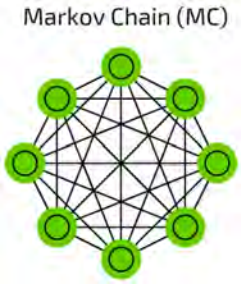



Fig. 2.12. Modelo de cadena de Markov [53]

- Generativa antagónica *Generative Adversarial Network*: consta de una red generadora, que produce datos de prueba a partir de una entrada, y de una red discriminadora, que debe diferenciar entre la información introducida y la que ha sido generada [46].

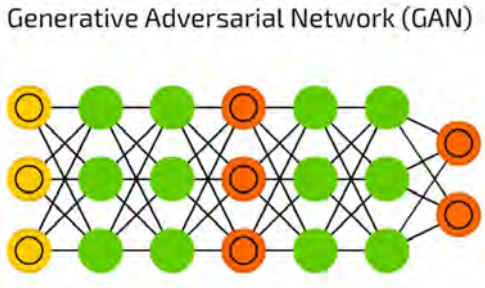



Fig. 2.13. Modelo de red generativa antagónica [53]

Al margen de esto se ha descubierto que, aumentando el número de capas intermedias dentro de una red neuronal, se produce una "jerarquía de características", mediante

la cual cada capa entrena unas determinadas propiedades en función de la salida obtenida de la anterior, con lo que se consigue una mayor complejidad y abstracción. Estas redes, a las que se les otorga el adjetivo de *profundas*, se utilizan especialmente en el análisis de grandes cantidades de datos no etiquetados o categorizados, como imágenes, texto o audio [55]. No obstante, presentan un importante inconveniente, ya que aumentar la profundidad puede conducir a una red sobreajustada (*overfitting*), donde se modelan los datos de entrenamiento demasiado bien, de tal forma que esta ha aprendido esa información de manera tan detallada que afecta negativamente a su desempeño con un nuevo conjunto; aunque también para este problema existen soluciones como el aumento de tamaño del set de entrenamiento, la adición de ruido o la finalización anticipada [56].

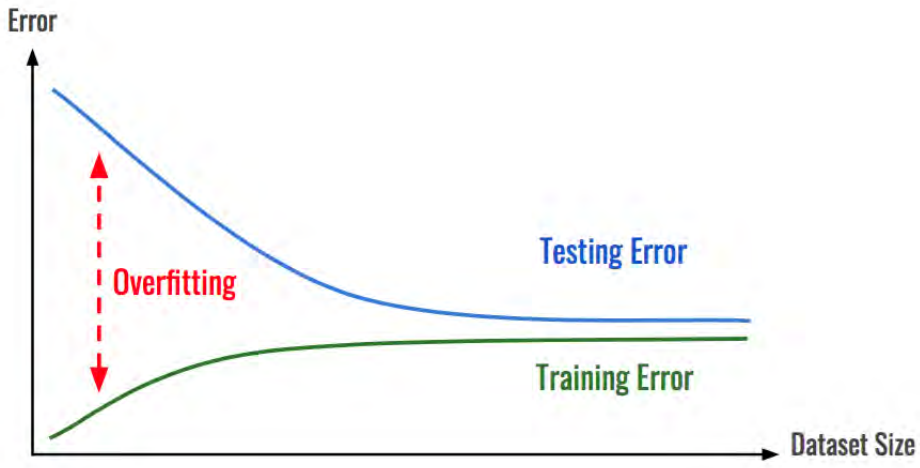



Fig. 2.14. Cuanto mayor sea el set de datos, menor probabilidad habrá de que se produzca un sobreajuste [57]

### 2.1.3. Técnicas de aprendizaje

En el aprendizaje de una red neuronal se diferencian dos metodologías según el tipo de datos con el que se alimente:

- Aprendizaje supervisado: se basa principalmente en "entrenar" la red mediante un conjunto de datos de entrenamiento, expresados de tal forma que a ciertas propiedades les corresponde una clasificación o etiqueta; para posteriormente introducir nuevos datos (de prueba) sobre los que se realizará una predicción con el objeto de ser clasificados [26]. Los problemas en los que se aplica el aprendizaje supervisado se pueden clasificar generalmente en:

- Clasificación: la variable de salida es una categoría, como especies de animales, colores, correo de spam... [58]
- Regresión: la variable de salida es un valor, como el precio de una casa, los ingresos de una persona o la puntuación que dará una persona a un restaurante [58].

| PassengerId | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 3 | Braund, Mr. Owen Harris | male | 22 | 1 | 0 | A/5 21171 | 7.25 | | S |
| 2 | 1 | 1 | Cumings, Mrs. John Bradley | female | 38 | 1 | 0 | PC 17599 | 712.833 | C85 | C |
| 3 | 1 | 3 | Heikkinen, Miss. Laina | female | 26 | 0 | 0 | 3101282 | 7.925 | | S |
| 4 | 1 | 1 | Futrelle, Mrs. Jacques Heath (Lily May Peel) | female | 35 | 1 | 0 | 113803 | 53.1 | C123 | S |
| 5 | 0 | 3 | Allen, Mr. William Henry | male | 35 | 0 | 0 | 373450 | 8.05 | | S |
| 6 | 0 | 3 | Moran, Mr. James | male | | 0 | 0 | 330877 | 84.583 | | Q |
| 7 | 0 | 1 | McCarthy, Mr. Timothy J | male | 54 | 0 | 0 | 17463 | 518.625 | E46 | S |

(a) Ejemplo de datos de entrenamiento.

| PassengerId | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked |
|---|---|---|---|---|---|---|---|---|---|---|
| 892 | 3 | Kelly, Mr. James | male | 34.5 | 0 | 0 | 330911 | 78.292 | B45 | Q |
| 893 | 3 | Wilkes, Mrs. James (Ellen Needs) | female | 47 | 1 | 0 | 363272 | 7 | | S |
| 894 | 2 | Myles, Mr. Thomas Francis | male | 62 | 0 | 0 | 240276 | 96.875 | E31 | Q |
| 895 | 3 | Wirz, Mr. Albert | male | 27 | 0 | 0 | 315154 | 86.625 | B57 B59 B63 B66 | S |
| 896 | 3 | Hirvonen, Mrs. Alexander (Helga E Lindqvist) | female | 22 | 1 | 1 | 3101298 | 122.875 | A21 | S |
| 897 | 3 | Svensson, Mr. Johan Cervin | male | 14 | 0 | 0 | 7538 | 9.225 | B36 | S |
| 898 | 3 | Connolly, Miss. Kate | female | 30 | 0 | 0 | 330972 | 76.292 | | Q |

(b) Ejemplo de datos de prueba. Nótese que no aparece la columna *Survived*, ya que el objetivo del problema consiste en analizar qué pasajeros era más probable que sobrevivieran.

Fig. 2.15. Extractos de los set de datos de entrenamiento (a) y prueba (b) de la competición *Titanic: Machine Learning from Disaster* [59] de *Kaggle*.

- Aprendizaje no supervisado: en este caso no existen datos de entrenamiento, por lo que no hay una clasificación, sino que lo que se busca es encontrar patrones en la información que describan de qué forma se estructura [26]. Los problemas en los que se aplica el aprendizaje no supervisado se pueden clasificar generalmente en:
  - Agrupamiento (*clustering*): consisten en agrupar los objetos en conjuntos según su similitud, ya sea para estudiar el comportamiento de los consumidores con fines de marketing, o para reconocer objetos dentro de una imagen [58].
  - Asociación: consisten en encontrar normas o reglas que describan grandes segmentos de datos, como la tendencia de personas que han adquirido un determinado artículo a comprar otro, o las similitudes en los rasgos de especies de animales [58].

Fig. 2.16. Muestra de imágenes del set de entrenamiento de la base de datos *STL-10* [60].

- Aprendizaje semisupervisado: comprende los problemas en los que una parte de los datos está etiquetada y el resto no, lo cual es bastante común en situaciones reales [46]. Por ejemplo, en imágenes de exámenes médicos un especialista puede marcar o clasificar ciertas anomalías, y mediante aprendizaje automático se puede evaluar el resto de pruebas, mejorando así tanto la eficiencia respecto a etiquetar todas las imágenes manualmente, como la precisión frente a un aprendizaje no supervisado [61].
- Aprendizaje reforzado: su funcionamiento se basa en prueba y error, alimentando la entrada según resulte correcta o incorrecta [62]. Tiene claras aplicaciones en la teoría de juegos o en economía.

## 2.2. Autoencoders

Como se ha indicado en apartados anteriores, los *autoencoders* o autocodificadores son un tipo de red neuronal que consiste en representar a la salida la entrada con la que haya sido alimentado. Para ello consta de un codificador (*encoder*), que comprime la entrada extrayendo las características que permitan su reconstrucción en el decodificador (*decoder*), el segundo segmento de la red.

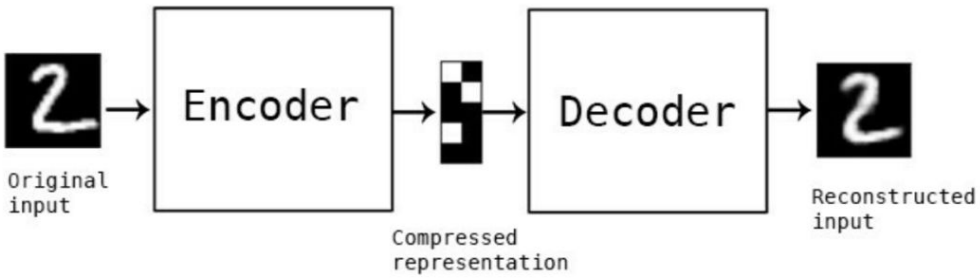



Fig. 2.17. Estructura de un *autoencoder* [63].

Generalmente, las capas ocultas suelen tener menos nodos que las de entrada y salida, de tal forma que se consigue una representación comprimida de los datos introducidos. No obstante, existen otras variantes de los *autoencoders*:

- *Variational autoencoders*: en este caso la red aprende los parámetros estadísticos que representan los datos, por lo que se pueden generar variaciones aleatorias de esa información [64] [65]. Se utilizan en aplicaciones como la generación de caras de personas basadas en famosos [66], o para crear pequeños fragmentos musicales [67].

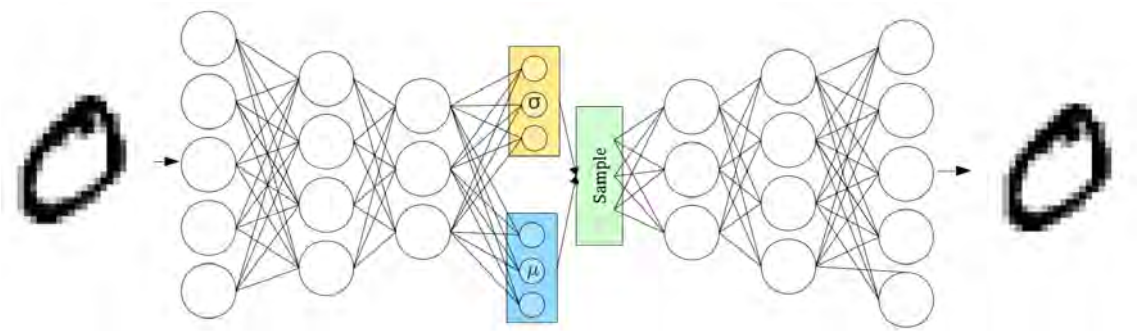
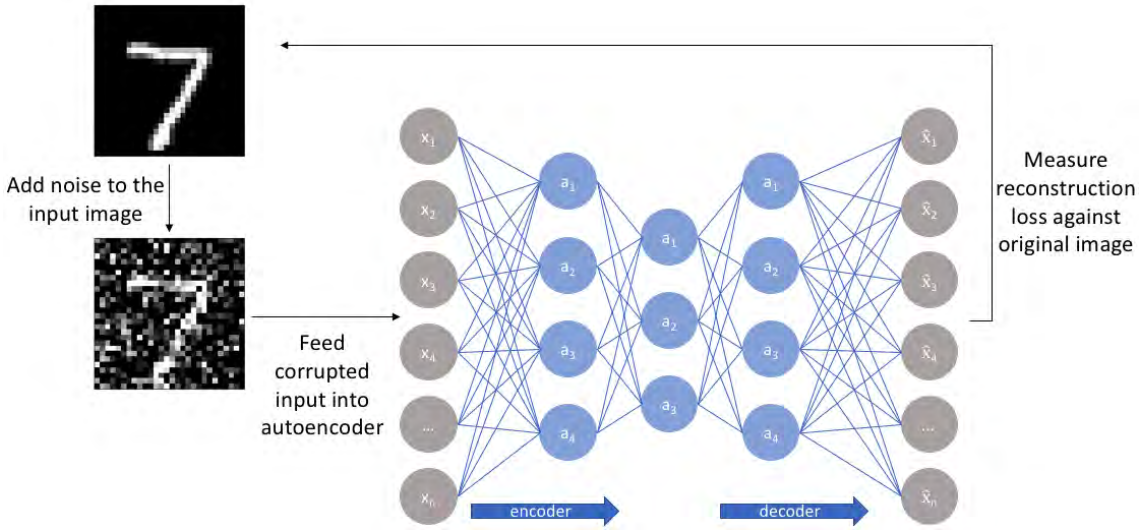
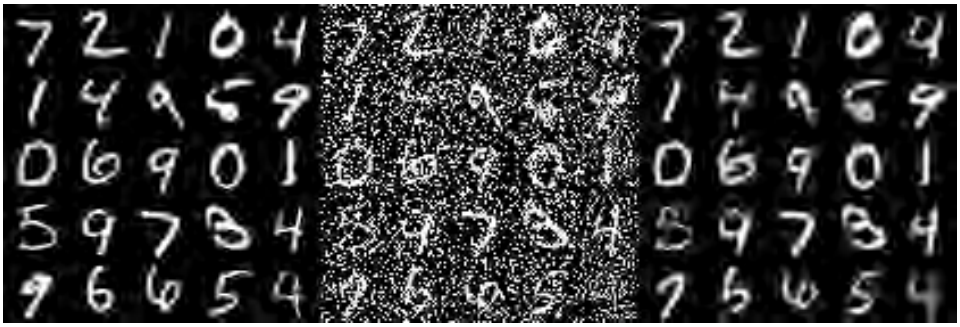



Fig. 2.18. Estructura de un *Variational autoencoder*. Nótese cómo, de la codificación, se obtienen vectores de parámetros estadísticos (desviación estándar, $\sigma$ , y media, $\mu$) [64]

- *Denoising autoencoders*: el funcionamiento de estas redes se basa en la adición de ruido a las entradas, para posteriormente entrenarla con el fin de recuperar la original sin alteraciones [68], como se puede observar en la figura 2.19.



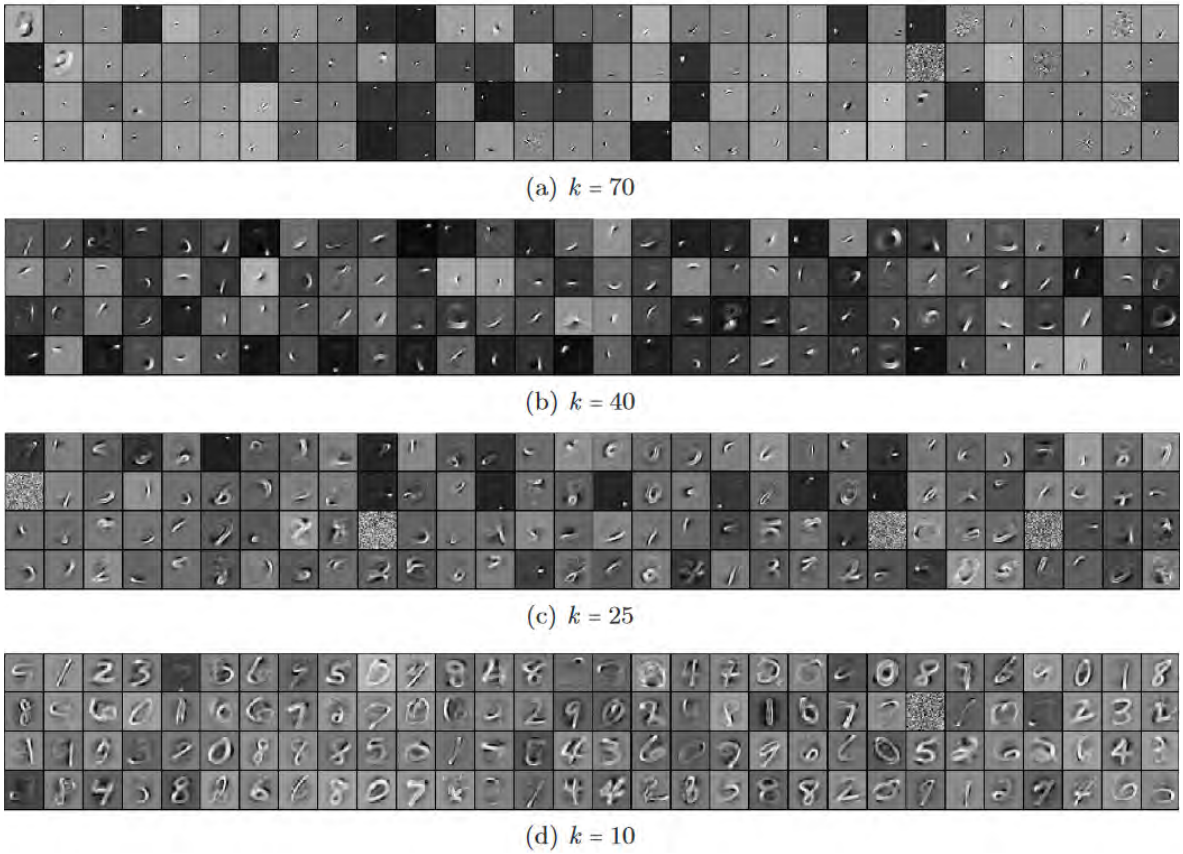

(a)



(b)

Fig. 2.19. (a) Funcionamiento de un *Denoising autoencoder* [69]. (b) Izquierda: set de imágenes originales. Centro: imágenes con ruido. Derecha: imágenes reconstruidas [70].

- *Sparse autoencoders*: se basan en la restricción de las activaciones de los nodos, de tal forma que están más "especializados" en características concretas [68]. Esta limitación se puede regular, dando lugar a los *k-sparse autoencoders*, donde *k* es el parámetro que indica la especialización de las neuronas [71] (figura 2.20).



(a) $k = 70$

(b) $k = 40$

(c) $k = 25$

(d) $k = 10$

Fig. 2.20. Filtros del *k-sparse autoencoder* para diferentes niveles de $k$ [71].

Debido a su capacidad para aprender y extraer rasgos y propiedades, los *autoencoders* se utilizan a menudo para "preentrenar" redes neuronales cuando no se posee una gran cantidad de datos de entrenamiento etiquetados [68], o para generar aleatoriamente nueva información similar a la entrada, como se ha visto anteriormente.

## 2.3. Herramientas para el desarrollo de redes neuronales

A continuación se presenta una serie de instrumentos y materiales, empleados para la programación y el aprendizaje de redes neuronales, indicando además cuáles son los más utilizados en el presente y el motivo.

### 2.3.1. Lenguajes de programación

- Python: es uno de los lenguajes de programación más populares, no sólo para los problemas relacionados con inteligencia artificial [9]. Esto se debe a que los comandos y algoritmos expresados son relativamente simples, lo que favorece su lectura;

pero principalmente a que cuenta con una gran extensión de librerías, que permiten realizar tareas de forma sencilla, sin escribir mucho código adicional. Se suele utilizar en plataformas de desarrollo como *Jupyter notebook* o *Spyder*, mediante distribuciones como *Anaconda*.

- R: es otro de los lenguajes más utilizados debido a su enfoque más estadístico, que permite realizar profundos análisis de datos de forma sencilla, combinándolos con una gran variedad de opciones para su visualización [72]. Al igual que Python, se puede programar mediante *Anaconda* en *Jupyter notebook* o *Spyder*.

- MATLAB: es una opción menos habitual debido a la necesidad de una licencia, a diferencia de las dos anteriores, que son totalmente gratuitas. La sencillez de su código hace que se emplee a menudo para introducirse en el cálculo y la estadística, y además cuenta con una gran variedad de librerías y herramientas para *data science* [73]. En este caso el lenguaje cuenta con software propio, MATLAB.

- SQL: se emplea para el manejo de bases de datos, sobre las que puede realizar operaciones de selección, inserción u organización [74]. Al margen de ser fácil de aprender, es muy popular al poder integrarse con otros lenguajes. Se puede utilizar en gestores de bases de datos como MySQL, Microsoft's SQL Server, Oracle DBM.

### 2.3.2. Librerías / *Frameworks*

Se presenta a continuación una serie de herramientas de uso característico en tareas relacionadas con *deep learning*.

- *TensorFlow*: se trata de una librería muy popular, desarrollada por Google, utilizada para llevar a cabo cálculos numéricos mediante diagramas de flujo de datos, y especializada en el aprendizaje automático [10]. Se puede ejecutar sobre una o varias CPU o GPU, lo que reduce notablemente el tiempo de ejecución.

- *Theano*: es una librería más antigua que *TensorFlow*, lo que la hace extensamente más documentada y, por tanto, utilizada. A pesar de esto, sus desarrolladores anunciaron en 2017 que dejarían de implementar nuevas características [75], lo que lleva a pensar que su uso irá decreciendo en el futuro.

- *Keras*: este integrador puede trabajar sobre *TensorFlow* o *Theano* (como *backend*)

mediante una interfaz muy intuitiva, lo que ha hecho que aumente considerablemente su empleo, especialmente para la programación de redes neuronales [76].

En contraste con los anteriores ítems, los que siguen son de uso común en cuestiones vinculadas al *Big Data*.

- *Apache Hadoop*: es la principal opción alternativa a *Spark*, ya que resulta más eficiente al operar con grandes cantidades de datos o aplicaciones pesadas, y mucho más segura [77].
- *Apache Spark*: es un *framework* de utilización muy extendida debido a su rapidez y facilidad de uso, al contar con sencillas interfaces para, entre otros, Python o Java [77].

La utilización de la siguiente herramienta, por contra, no se restringe a problemas concretos, sino que se considera de uso general.

- *Elephas*: supone una extensión de *Keras* que actúa como integrador, permitiendo manejar modelos distribuidos de *Deep Learning* con *Spark* [13].

## 2.4. Líneas de trabajo actuales

Las redes neuronales están muy presentes en la actualidad en múltiples ámbitos de la vida diaria. Por poner algunos ejemplos muy comunes, plataformas de vídeo bajo demanda, como es el caso de Netflix, utilizan sistemas de recomendación de contenido basados en el aprendizaje profundo [78]. De igual forma, numerosas tiendas por internet emplean algoritmos similares para recomendar artículos que otros clientes han comprado, o algunas páginas web para ofrecer anuncios personalizados. Por otro lado, las redes sociales emplean redes neuronales, principalmente convolucionales, para el reconocimiento, no sólo facial, sino también de objetos, con el fin de ofrecer etiquetas para añadir a la foto en cuestión. Además, los asistentes personales, presentes en aparatos inteligentes, desde teléfonos hasta televisores, utilizan técnicas como las que se han indicado en apartados anteriores para reconocer la voz de la persona que les está hablando y llevar a cabo las tareas señaladas. A continuación se ilustran algunos de los ejemplos planteados:


Fig. 2.21. Ejemplo del sistema de recomendación de Netflix.

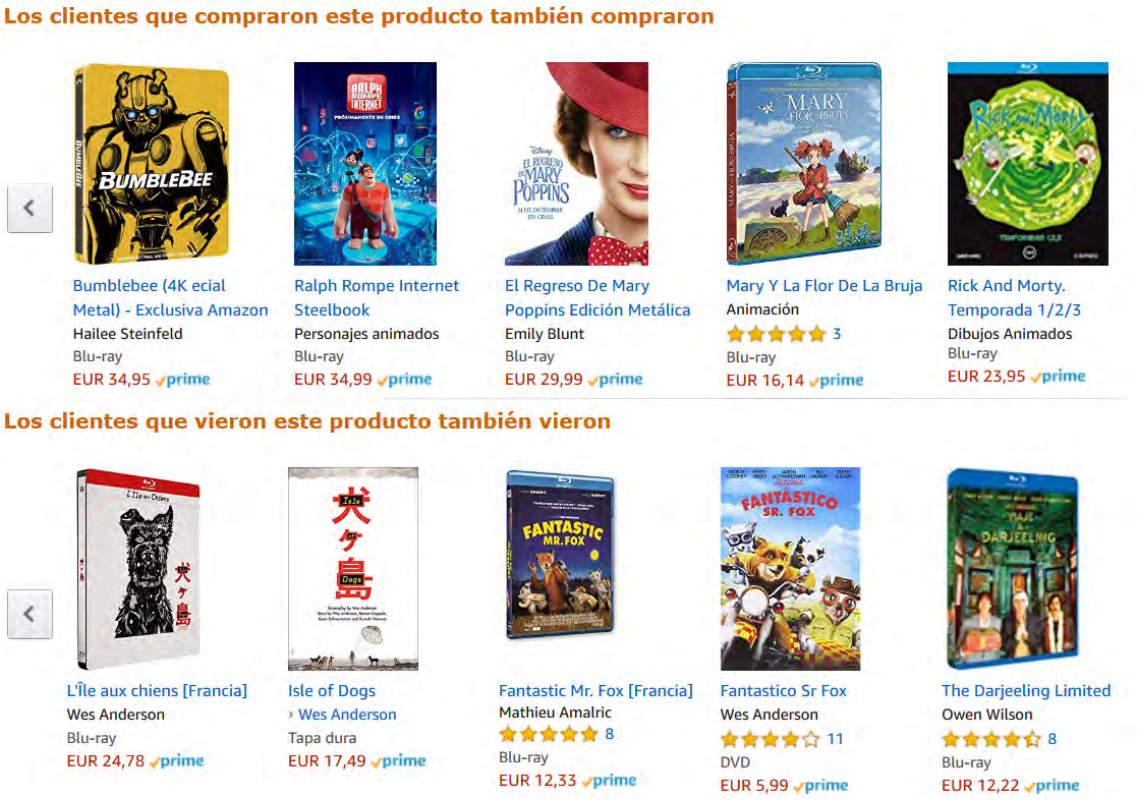



Fig. 2.22. Ejemplo del sistema de recomendación de Amazon.

Al margen de estos casos de uso prácticos, la inteligencia artificial y, más concretamente, el aprendizaje profundo o las redes neuronales tienen aplicaciones en campos tan variados como la navegación GPS, la videovigilancia, las búsquedas en internet, la traducción simultánea, la detección de fraude, la teoría de juegos o el filtro de correo basura o virus informáticos.

### 2.4.1. Casos actuales y previstos

En un mundo tan conectado como el que se ha presentado, el futuro de las redes neuronales y del análisis de datos en definitiva, deberá ir estrechamente ligado a normativas de protección de la información ya que, de no ser así, la seguridad y privacidad de la

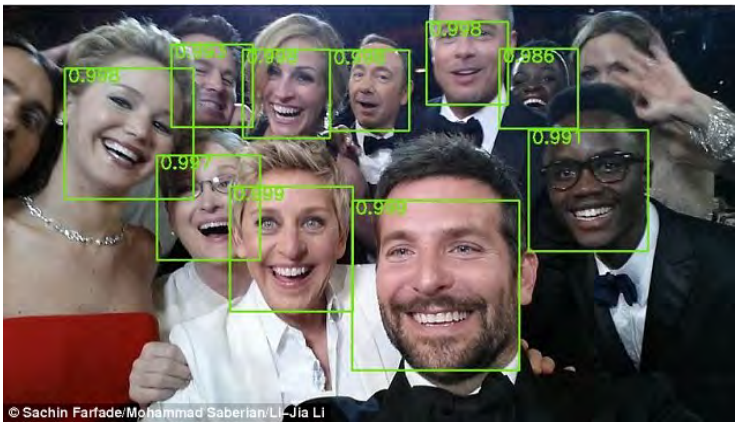



Fig. 2.23. Ejemplo de reconocimiento facial en redes sociales [79].

población estaría en juego, con los correspondientes riesgos que ello conlleva. Se puede encontrar más información sobre esto en el anexo A.

Además, es de esperar que el ámbito de aplicación del *deep learning* y sus ramas se extienda, lo cual está empezando a suceder actualmente, a sectores como la educación, la medicina, el transporte o la industria, revolucionándolos totalmente en el proceso:

- Educación: se puede aplicar al estudio de la trayectoria académica de los alumnos, prediciendo su futuro profesional o personalizando su forma de estudiar; así como a la detección de plagios, o a la customización y organización de los contenidos impartidos.

- Medicina y salud: en este entorno se espera un gran despliegue de inteligencia artificial para el diagnóstico, la identificación y detección de enfermedades, basados en imágenes, haciendo estos procesos más rápidos, precoces y precisos; gracias también al gran volumen de datos de casos similares o nuevas técnicas del que disponen. Además, debido a la información que se maneja de los pacientes, se puede ofrecer una atención personalizada, determinando con mayor eficacia el tiempo que necesitan estar ingresados en el hospital y reduciendo la cantidad de reingresos, evitando así gastos prescindibles en salud que podrían destinarse a la investigación y el desarrollo de nuevos tratamientos. Cabe destacar también el caso de la genética, cuya evolución podría venir definida por las aplicaciones del análisis de datos en el estudio de los genes, su identificación y predicción.

- Economía y finanzas: en este sector ya se han producido avances relacionados con el *data science*, prueba de ello es el surgimiento de las *fintech*, empresas que utilizan la tecnología para mejorar los servicios financieros; o las criptomonedas. Debido a la

gran cantidad de información almacenada relacionada con transacciones, clientes, transferencias, ingresos... la inteligencia artificial se puede aplicar para la automatización de procesos, con lo que se aumentaría considerablemente la productividad y la experiencia del usuario; la seguridad en las operaciones y la planificación de estrategias bursátiles y de cotización.

- Fuerzas armadas: en la "guerra electrónica" se ha invertido una gran cantidad de fondos para el desarrollo de sistemas, como la detección de patrones en huellas o señales que indiquen el origen amistoso u hostil de las mismas, la identificación de objetos en tiempo real, el reconocimiento facial de objetivos, el entrenamiento y la simulación de combate. Además, el uso de drones, por control remoto o mediante rutas predefinidas, se ha extendido a labores de seguridad, patrulla o identificación e incluso eliminación de personas, con el fin de evitar pérdidas humanas.

- Transporte: la aparición en los últimos años de los coches autónomos lleva consigo una fuerte componente de *data science* en cuanto a reconocimiento en tiempo real de señales y otros objetos y al control de la seguridad de los ocupantes y del propio vehículo. Resulta evidente, por tanto, que en el futuro se verá aplicada a casos más complejos como la monitorización y regulación de los sistemas de transporte y de tráfico, tanto terrestre como aéreo, actuando diligentemente en caso de accidentes e imprevistos; el análisis predictivo del uso de transporte público o la detección de averías y fallos en los vehículos. Cabe destacar también las aplicaciones que la inteligencia artificial tendrá en la exploración espacial, en la transmisión de datos, su compresión y posterior filtrado; en el análisis del espacio y de los planetas para predecir e investigar anomalías y fenómenos como el viento solar, los agujeros negros...; en la navegación espacial no tripulada, en el estudio del despegue de cohetes y, en definitiva, de la búsqueda de vida.

- Comunicaciones: además de los casos ya planteados de filtrado de ruido y compresión en señales, y de asistentes de voz o *chatbots*; existen otras aplicaciones futuras como la generación de texto a partir de palabras clave o la detección de patrones en la localización de los usuarios, con lo que se podrían tomar medidas como la reorientación de antenas.

- *Big Data*: como ya se ha mencionado anteriormente en el trabajo, este concepto hace referencia a un volumen de datos que por su complejidad o tamaño resultan inabordables para los algoritmos de análisis tradicionales, ya sea por su obtención,

su almacenaje, su ordenación o su transporte. Esta idea, que claramente interseca con el resto de industrias mencionadas, está relacionada a su vez con otras dos, la tecnología 5G y el *internet de las cosas* o *Internet of Things*, *IoT*:

  - 5G: la quinta generación de telefonía móvil llegará requerida por un enorme crecimiento en la demanda de ancho de banda, bien para actividades que necesitan mucho, como la visualización de vídeo en *streaming* u otros servicios de abundantes contenidos; o bien en pequeñas cantidades para la comunicación entre máquinas, como se explica en el siguiente punto. De la misma forma, la constante generación de datos desde los distintos dispositivos conforma una importante fuente de información
  - Internet de las cosas: está estrechamente relacionado tanto con el *Big Data*, como con el 5G, y consiste en una extensa red de dispositivos inteligentes y sensores conectados entre sí.

- Domótica y ciudades inteligentes (*smart cities*): se relacionan especialmente con el internet de las cosas en el sentido de que implementan sistemas de automatización para facilitar la vida de los habitantes, mediante la conexión total de sensores y aparatos dentro y fuera del hogar. En lo relacionado con el ámbito doméstico se pueden automatizar procesos como la regulación de persianas, iluminación, temperatura o dispositivos de riego en función de las circunstancias meteorológicas; el ajuste de aparatos como altavoces, repetidores de señal Wi-Fi, luces o puertas inteligentes en función de la ubicación de los residentes. Estas técnicas se pueden escalar a las ciudades en situaciones como la regulación del tráfico según las necesidades, la sostenibilidad mediante la instalación de paneles fotovoltaicos orientables, molinos eólicos o la adecuación de la intensidad del alumbrado público, así como la gestión eficiente de agua y residuos. Además de esto, la integración de nuevas tecnologías y dispositivos como *chatbots*, pantallas interactivas, drones, 5G, robótica o *blockchain*.

En cada uno de los ámbitos previamente mencionados ya se trabaja en algunos casos de uso, como los coches autónomos, la automatización de servicios y acciones en el hogar y en la industria, o las criptodivisas; siendo su relevancia actual tal que es lógico esperar un importante desarrollo a corto plazo. Además, todos los ámbitos se verán influenciados en el futuro por los últimos avances en tecnología, como la computación cuántica, los robots (RPA: *Robotic Process Automation* o automatización robótica de procesos) o los

ya mencionados *Big Data* y 5G.

## 2.5. Crítica al estado del arte

Uno de los principales problemas a los que se enfrenta el análisis de datos en la actualidad fue definido en la década de 1960 por Gene Amdahl en la ley que lleva su nombre, y reside en la susceptibilidad de algunos algoritmos de *deep learning* a esta formulación:

$$A = \frac{1}{(1 - F_m) + \frac{F_m}{A_m}} \tag{2.1}$$

Donde $A$ hace referencia al aumento en velocidad obtenido en un sistema al mejorar uno de sus subsistemas, $A_m$, al factor de mejora de dicho subsistema, y $F_m$, al tiempo en que éste ha estado activo. Se puede deducir, por tanto, de las anteriores indicaciones, que la ley de Amdahl [80] hace referencia a las ventajas en tiempo obtenidas debido al incremento en el rendimiento de uno de varios procesadores en un caso de paralelización.

Tal y como se deriva del análisis matemático de la ecuación, la ley de Amdahl presenta gráficamente una asíntota, de forma que el incremento en la aceleración se estanca independientemente de cuantos procesadores se añadan:

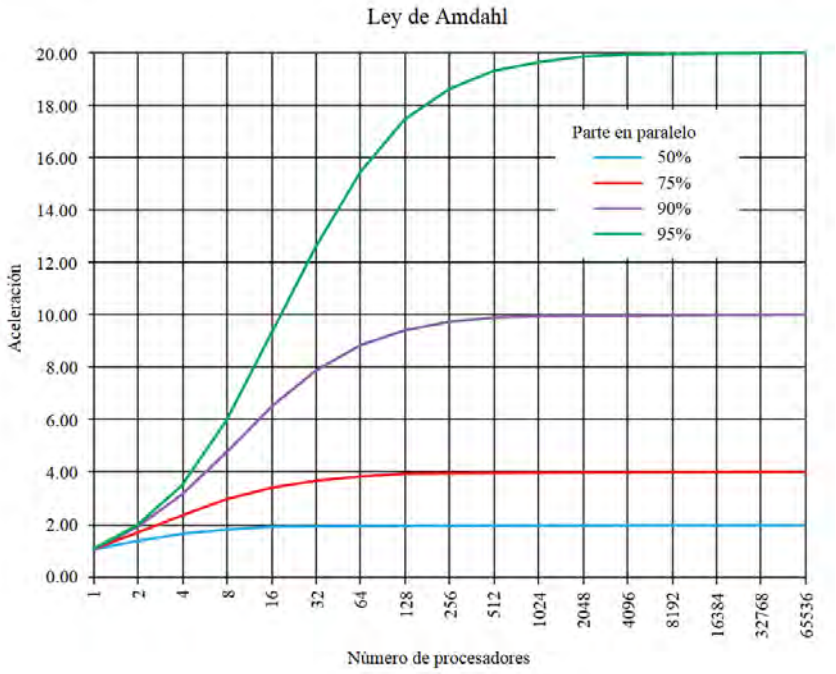



Fig. 2.24. Ejemplo de aumento en la aceleración según la ley de Amdahl [81].

En comparación con la de Amdahl, que basaba su aplicación a casos en los que el

tamaño del problema se mantenía, la ley de Gustafson ofrece una visión más optimista del caso de paralelización, centrándose en mantener el tiempo de ejecución cuando aumenta la carga de trabajo, lo que se puede conseguir mediante un incremento en el número de procesadores. Estas dos formas de afrontar el tema de la escalabilidad dan lugar a dos conceptos, escalado fuerte y débil, correspondientes a los casos descritos por Amdahl y Gustafson respectivamente [82].

$$A(P) = P - \alpha \cdot (P - 1) \tag{2.2}$$

Con $\alpha = a/(a + b)$, donde $A$ es la aceleración, $P$ el número de procesadores, $a$ el tiempo secuencial, y $b$ el paralelo.

CAPÍTULO 3

# ANÁLISIS DEL PROBLEMA

Como se planteó al final del apartado anterior, un aumento considerable en el volumen de datos manejado en un problema, como sucede frecuentemente en los relacionados con entornos *Big Data*, puede suponer una importante penalización en su tiempo de ejecución. Para contrarrestar este inconveniente, tal y como indica la ley de Gustafson, se puede incrementar el número de procesadores dedicados hasta un límite teórico y práctico, con el objetivo de que trabajen de forma paralela. Además de ello, en la actualidad existe una serie de herramientas, descritas anteriormente, que se pueden emplear, a veces de forma conjunta, en estos problemas para obtener una mayor eficiencia. Es por esto que en el presente trabajo se evaluará su comportamiento, en primer lugar de forma individual, y posteriormente coordinada; principalmente en cuanto a cómo afecta su adición al rendimiento del código, así como al error obtenido de los resultados del mismo. Teniendo en cuenta este fin se propone un objetivo principal, así como tres más específicos:

- Como meta general está la evaluación del rendimiento, además de la degradación, tanto en tiempo como en precisión, producida en el entrenamiento distribuido de algoritmos de *Deep Learning*, en este caso *autoencoders*, en entornos *Big Data*.
- Establecer una base comparativa de las distintas herramientas de *Deep Learning* en diferentes entornos y problemas.
- Evaluar el *overhead*, es decir, los sobrecostes debidos a la inclusión de capas de abstracción, que aunque permiten una mejor programación, llevan asociados un coste de ejecución.

- Analizar factores adicionales, como la parte de comunicaciones, el balance entre costes y beneficios en la paralelización, la degradación de los algoritmos de entrenamiento en su versión distribuida, o la degradación de los tiempos a partir de la cantidad de datos manejados.

Al margen de esto, es relevante destacar que, debido a que los algoritmos, librerías y bases de datos empleados son de uso común, y los experimentos fácilmente reproducibles, todos estos objetivos vendrán apoyados por referencias bibliográficas pertenecientes a la literatura habitual del tema.

CAPÍTULO 4

# DISEÑO E IMPLEMENTACIÓN DE LA SOLUCIÓN

## 4.1. Introducción

A continuación se plantea un conjunto de resoluciones al problema planteado en el anterior capítulo. Lo que se pretende con ello es ofrecer un análisis comparativo en cuanto a rendimiento de, como ya se adelantó previamente en su presentación, varias de las herramientas más empleadas en tareas de análisis de datos, concretamente de *TensorFlow*, *Theano*, *Keras* y *Spark*, en cuya implementación del problema de los autoencoders se profundizará en esta sección. Cabe recalcar que el código completo, así como una explicación de la metodología de lanzamiento, se puede encontrar en la documentación perteneciente al anexo.

### 4.1.1. Características comunes a las soluciones

Reincidiendo en la descripción ofrecida anteriormente, los autoencoders son un tipo de red neuronal cuyo funcionamiento radica en replicar a su salida los datos introducidos de manera casi exacta. Suele ser en ellos común, por tanto, que el número de nodos de sus capas sea simétrico con respecto a la que se encuentre en el centro del modelo [69]. Según sea el tamaño de las capas intermedias se conseguirán reducir las características representadas, conservando sólo las más representativas, en el caso de que sea menor; o aumentar el número de esos parámetros, con el fin de definir mejor el conjunto de datos,

en el caso contrario. En esta situación se propone un modelo, planteado en el artículo [83], donde se evalúa una metodología óptima para las bases de datos a utilizar en este trabajo; que intenta combinar ambas ideas, ensanchando las capas primero, con lo que se consigue representar un mayor número de propiedades, para a continuación estrecharlas, conservando únicamente las que mejor representen el modelo, quedando de la siguiente manera:

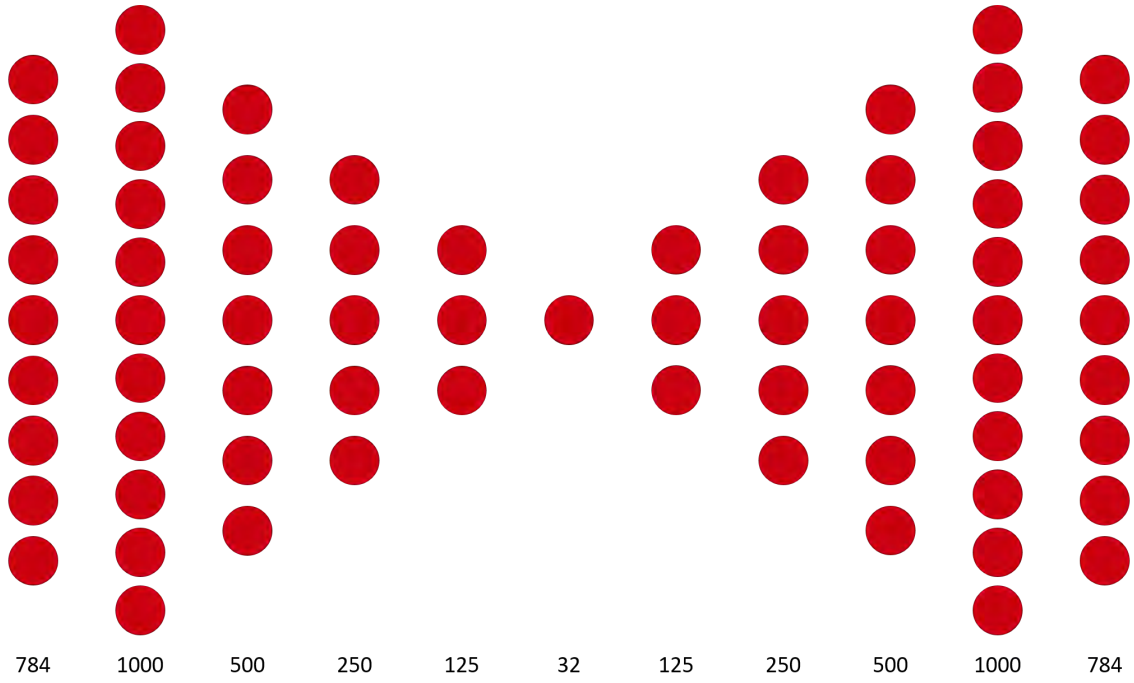



Fig. 4.1. Modelo de capas propuesto con número de nodos por cada una.

Dentro de la red neuronal existe una serie de hiperparámetros, cuyos valores, inicialización y actualización se han elegido para que sean iguales para todos los casos, y se describirán a continuación. En primer lugar se suelen definir los valores iniciales de, tanto los pesos, como los *bias*, los cuales es habitual que procedan de una distribución aleatoria, ya sea uniforme, normal... y cuyos parámetros dependerán del tipo de inicialización escogido. Estos valores, que normalmente se originan de la misma manera para todas las capas, varían a través de las épocas, esto es, de cada iteración completa del conjunto de datos de entrenamiento; de la forma definida por el optimizador, y regulada por las distintas variables con que se le alimentan, entre las cuales destaca principalmente la tasa de aprendizaje. Esta supone, a grandes rasgos, un controlador del grado con el que varían los pesos y los *bias*, o de forma equivalente, la velocidad a la que el modelo aprende, lo que, llevado a los extremos (su valor suele estar comprendido entre 0 y 1) puede hacer que los pesos oscilen demasiado, produciendo el mismo efecto en las pérdidas; o en el caso de que

la tasa sea muy pequeña, estas no lleguen a converger a lo largo de las épocas. Además de esto, los nodos de cada capa de la red llevan definidos una función de activación, de la que dependerá que estos sean estimulados ante ciertos datos de entrada, lo cual repercutirá, por tanto, en el proceso de entrenamiento. Por último, cabe resaltar que el entrenamiento se produce por bloques, cuyo tamaño supone un importante parámetro a definir, ya que indica el número de muestras con que se entrenará la red dentro de una misma época. De nuevo, esta variable puede llevarse a los dos extremos: un tamaño de una única muestra podría optimizar los pesos y *bias* en una dirección no deseada, suponiendo, no obstante, un menor coste computacional; mientras que en el otro límite, en el que la capacidad del bloque comprende todo el conjunto de datos, no se corre el riesgo de que la optimización avance erróneamente, aunque computacionalmente el entrenamiento sería más costoso. La clave en la elección de este parámetro será, por tanto, oscilar entre ambos extremos para encontrar la solución que mejor se adapte a las circunstancias del problema.

## 4.2. TensorFlow

A continuación se describirá la solución desarrollada mediante *TensorFlow* en el lenguaje de programación *Python*:

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import math
```

En primer lugar se importan las librerías a utilizar: *TensorFlow*, *NumPy* y *Math*. *Keras* se añade únicamente para cargar la base de datos.

```
(train_images, _), (test_images, _) = keras.datasets.fashion_mnist.
    load_data()
train_images = train_images / 255.0
test_images = test_images / 255.0
```

A continuación se almacenan de la base de datos, ya sea *Fashion MNIST* o *MNIST*, únicamente las imágenes, ya que al no tratarse de un problema de clasificación, las eti-

quetas no serán necesarias. Posteriormente se normalizan los datos de entrenamiento y de prueba a un valor máximo de 1, dividiendo todos entre el mayor de los mismos, que será 255. Esto se realiza para que las diferencias entre los valores sean más pequeñas, y así por ejemplo una variación en los pesos de la red neuronal no otorgue mayor influencia en ella a los dígitos más altos.

```
n = [784, 1000, 500, 250, 125, 32, 125, 250, 500, 1000, 784]

weights = {
    'l1': tf.Variable(tf.random_uniform([n[0], n[1]])),
    'l2': tf.Variable(tf.random_uniform([n[1], n[2]])),
    'l3': tf.Variable(tf.random_uniform([n[2], n[3]])),
    'l4': tf.Variable(tf.random_uniform([n[3], n[4]])),
    'l5': tf.Variable(tf.random_uniform([n[4], n[5]])),
    'l6': tf.Variable(tf.random_uniform([n[5], n[6]])),
    'l7': tf.Variable(tf.random_uniform([n[6], n[7]])),
    'l8': tf.Variable(tf.random_uniform([n[7], n[8]])),
    'l9': tf.Variable(tf.random_uniform([n[8], n[9]])),
    'out': tf.Variable(tf.random_uniform([n[9], n[10]]))
}

biases = {
    'l1': tf.Variable(tf.zeros([n[1]]))
    'l2': tf.Variable(tf.zeros([n[2]]))
    'l3': tf.Variable(tf.zeros([n[3]]))
    'l4': tf.Variable(tf.zeros([n[4]]))
    'l5': tf.Variable(tf.zeros([n[5]]))
    'l6': tf.Variable(tf.zeros([n[6]]))
    'l7': tf.Variable(tf.zeros([n[7]]))
    'l8': tf.Variable(tf.zeros([n[8]]))
    'l9': tf.Variable(tf.zeros([n[9]]))
    'out': tf.Variable(tf.zeros([n[10]]))
}
```

Después se define la arquitectura de la red mediante la construcción de un *array* que contiene el número de nodos por cada capa, y que se utilizará más adelante; así como dos diccionarios, que almacenan los pesos y los *bias*. En el primer caso, los valores se inicializan mediante la metodología definida, y para cada capa tienen un tamaño de

$n_{entradas} \times n_{salidas}$. Por otro lado, a los *bias* se les da un valor inicial, y se almacenan en su propio diccionario con un tamaño de $1 \times n_{salidas}$.

```
def autoencoder(layer_in, weights, biases):
    layer_1 = tf.nn.relu(tf.add(tf.matmul(layer_in, weights['l1']),
        biases['l1']))
    layer_2 = tf.nn.relu(tf.add(tf.matmul(layer_1, weights['l2']),
        biases['l2']))
    layer_3 = tf.nn.relu(tf.add(tf.matmul(layer_2, weights['l3']),
        biases['l3']))
    layer_4 = tf.nn.relu(tf.add(tf.matmul(layer_3, weights['l4']),
        biases['l4']))
    layer_5 = tf.nn.relu(tf.add(tf.matmul(layer_4, weights['l5']),
        biases['l5']))
    layer_6 = tf.nn.relu(tf.add(tf.matmul(layer_5, weights['l6']),
        biases['l6']))
    layer_7 = tf.nn.relu(tf.add(tf.matmul(layer_6, weights['l7']),
        biases['l7']))
    layer_8 = tf.nn.relu(tf.add(tf.matmul(layer_7, weights['l8']),
        biases['l8']))
    layer_9 = tf.nn.relu(tf.add(tf.matmul(layer_8, weights['l9']),
        biases['l9']))
    layer_out = tf.nn.sigmoid(tf.add(tf.matmul(layer_9, weights['out
        ']), biases['out']))
    return layer_out
```

Posteriormente se define la función *autoencoder*, a la que se introducirán como parámetros ambos diccionarios previamente definidos, así como un marcador o *placeholder* de la capa de entrada; y que retornará la de salida. Dentro de ella se definen las funciones de activación utilizadas, además de unir las capas mediante la multiplicación de las entradas a cada una por los pesos de la misma, y su posterior suma con sus *bias*.

```
epochs = 720
batch_size = 500
layer_in = tf.placeholder("float", [None, n[0]])
out = tf.placeholder("float", [None, n[10]])
cost = tf.reduce_mean(tf.square(layer_out - out))
optimizer = tf.train.AdamOptimizer(learning_rate=0.001).minimize(
```

```
    cost)
predictions = autoencoder(layer_in, weights, biases)
```

Por último, antes de proceder al entrenamiento de la red, se determinan los parámetros restantes, como el número de épocas y el tamaño del bloque de entrenamiento o *batch*, descritos anteriormente. Además de esto, se declaran los marcadores de la capa de entrada y de salida, con un tamaño de 784 columnas y número de filas indeterminado, ya que dependerá del valor del bloque de entrenamiento. La función de coste se define en el caso de los *autoencoders* como el error cuadrático medio entre la salida de la red y el objetivo, que a su vez es la misma entrada con que ha sido alimentada la misma; y será minimizada mediante el optimizador seleccionado, dependiendo de la tasa de aprendizaje con que haya sido alimentado.

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch in range(epochs):
        avg_cost = 0.0
        batch_number = int(len(train_images) / batch_size)
        train_images_batches = np.array_split(train_images,
            batch_number)
        for i in range(batch_number):
            batch_images = train_images_batches[i]
            _,c = sess.run([optimizer, cost],feed_dict={
                layer_in: batch_images,
                out: batch_images
                })
            avg_cost += c/batch_number
        print('Epoch', epoch, '/', hm_epochs, 'loss:', epoch_loss)
```

Para llevar a cabo la ejecución se debe iniciar una sesión de *TensorFlow* en la que, en primer lugar, se inicializan las variables, para a continuación proceder al entrenamiento de la red mediante dos bucles *for* anidados. Estos recorrerán completamente un número de veces igual a *epochs* el conjunto de los datos, que será dividido en bloques de tamaño *batch_size*, alimentados a la red neuronal para su aprendizaje. Además de esto cabe destacar que, en concordancia con el concepto de *autoencoder*, la información que se introduce en la capa de entrada, así como en el objetivo a conseguir, es la misma. De esta forma, como ya se indicó en párrafos anteriores, las pérdidas se calculan para cada bloque, y sus

valores se acumulan con el fin de hallar su media, y ofrecer el valor total perteneciente a cada época.

## 4.3. Theano

El diseño de la solución para *Theano* será equivalente al desarrollado para *TensorFlow*, y se describe paso a paso a continuación:

```
import theano
import theano.tensor as T
import numpy as np
from keras.datasets import fashion_mnist
from lasagne.updates import adam
```

En primer lugar se importan las librerías a utilizar: *Theano*, *NumPy* y *Lasagne*. *Keras* se añade, de nuevo, únicamente para cargar la base de datos.

```
class Layer(object):
    def __init__(self, input, n_in, n_out, W=None, b=None,
        activation_fn=None):
        self.input = input
    if W is None:
        W_values = np.random.uniform(low=-np.sqrt(6./(n_in)),
                        high=np.sqrt(6./(n_in)),
                        size=(n_out, n_in)).astype(theano.config.
                            floatX)
        W = theano.shared(name='Weights', value=W_values, borrow=
            True)
    if b is None:
        b_values = np.zeros(n_out, dtype=theano.config.floatX)
        b = theano.shared(name='bias', value=b_values, borrow=True)
    self.W = W
    self.b = b
    l_output = (T.dot(self.W, input.T)).T + self.b
    self.output = activation_fn(l_output)
    self.params = [self.W, self.b]
```

A continuación se crea la clase *Layer* con su constructor, en la que se inicializan los valores de los pesos según la metología elegida, así como los de los *bias*, de la misma forma en la que se hacía con *TensorFlow*.

```
input = T.matrix(name="input", dtype=theano.config.floatinput)
lr = T.scalar(name='learning_rate', dtype=theano.config.floatinput)

in_layer = Layer(input, 784, 1000, activation_fn=T.nnet.relu)
enc_layer_1 = Layer(in_layer.output, 1000, 500, activation_fn=T.nnet
    .relu)
enc_layer_2 = Layer(enc_layer_1.output, 500, 250, activation_fn=T.
    nnet.relu)
enc_layer_3 = Layer(enc_layer_2.output, 250, 125, activation_fn=T.
    nnet.relu)
enc_layer_4 = Layer(enc_layer_3.output, 125, 32, activation_fn=T.
    nnet.relu)
dec_layer_1 = Layer(enc_layer_4.output, 32, 125, activation_fn=T.
    nnet.relu)
dec_layer_2 = Layer(dec_layer_1.output, 125, 250, activation_fn=T.
    nnet.relu)
dec_layer_3 = Layer(dec_layer_2.output, 250, 500, activation_fn=T.
    nnet.relu)
dec_layer_4 = Layer(dec_layer_3.output, 500, 1000, activation_fn=T.
    nnet.relu)
out_layer = Layer(dec_layer_4.output, 1000, 784, activation_fn=T.
    nnet.sigmoid)

params = in_layer.params + enc_layer_1.params + enc_layer_2.params +
     enc_layer_3.params + enc_layer_4.params + dec_layer_1.params +
    dec_layer_2.params + dec_layer_3.params + dec_layer_4.params +
    out_layer.params
```

Posteriormente se crean dos tensores, uno para la entrada de datos y otro para la tasa de aprendizaje, a los que se introducirán valores más adelante. Además de esto, se construye la red con el número de capas y nodos especificados anteriormente, y se asignan las funciones de activación dependiendo del experimento que se esté realizando.

```
params = in_layer.params + enc_layer_1.params + enc_layer_2.params +
```

```
     enc_layer_3.params + enc_layer_4.params + dec_layer_1.params +
    dec_layer_2.params + dec_layer_3.params + dec_layer_4.params +
    out_layer.params
cost = 0.5 * T.mean((input-out_layer.output)**2)
grads = T.grad(cost, wrt=params)
updates = adam(grads, params, learning_rate=0.001)
f = theano.function(inputs=[input, lr], outputs=[cost, lr], updates=
    updates, allow_input_downcast=True)
```

Más adelante se define la función de coste, que consistirá en el error cuadrático medio entre la entrada y la salida de la red, al tratarse de un *autoencoder*; y el optimizador seleccionado para el experimento en cuestión. También cabe destacar que en el caso de este último se introduce la tasa de aprendizaje elegida. Asimismo, se declara una función de *Theano* en la que se introducen los datos de entrada al modelo, y de la que se extrae el resultado correspondiente al cálculo de la función de costes.

```
def train(training_data, learning_rate=0.001, batch_size=240, epochs
    =1000):
     total_values = len(training_data)
     for epoch in range(epochs):
         np.random.shuffle(training_data)
         mini_batches = [training_data[k: k+batch_size] for k in
             range(0, total_values, batch_size)]
         training_cost = 0
         for mini_batch in mini_batches:
             cost_ij , _ = f(mini_batch, learning_rate)
             training_cost += cost_ij
```

Finalmente se define la función *train*, a la que se introducen como argumentos los datos que se vayan a utilizar, la tasa de aprendizaje, el tamaño del bloque de entrenamiento y el número de épocas. Dentro de *train*, como ya pasaba en el caso de *TensorFlow*, tienen lugar dos bucles *for* anidados, que recorren primeramente el número de épocas, y dentro de cada una de ellas, el conjunto de datos de entrenamiento dividido en bloques, que se utilizan para calcular e ir acumulando los errores resultantes de la ejecución, con el fin de obtener el valor medio propio de cada época.

```
(x_train, _), (x_test, _) = fashion_mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (x_train.shape[0] , (x_train.shape[1]
    * x_train.shape[2])))
x_test = np.reshape(x_test, (x_test.shape[0] , (x_test.shape[1] *
    x_test.shape[2])))
train(x_train)
```

Por último, en el código principal del archivo se importan los conjuntos de datos de entrenamiento y prueba, y se realiza sobre ellos una normalización, inmediatamente anterior a alimentación como argumento a la función *train*, creada en el paso previo.

## 4.4. Keras

En este caso debe aclararse que el código de *Python* es el mismo tanto para el caso en que el *backend* es *TensorFlow*, como para cuando se trata de *Theano*, lo cual se especifica a la hora de lanzar el proceso. Como se comprobará a continuación, el hecho de que la programación de *Keras* se realice a mayor nivel, se traduce en un código más corto y fácilmente explicable.

```
import numpy as np
from keras.layers import Input, Dense
from keras.models import Model
from keras import optimizers
from keras.datasets import fashion_mnist
```

En primer lugar se importa la librería *NumPy*, así como diversas herramientas de *Keras* que serán utilizadas a lo largo de la ejecución.

```
input_layer = Input(shape=(784,))
enc_1 = Dense(1000, activation='relu', kernel_initializer='
    he_uniform')(input_layer)
enc_2 = Dense(500, activation='relu', kernel_initializer='he_uniform
    ')(enc_1)
```

```
enc_3 = Dense(250, activation='relu', kernel_initializer='he_uniform
    ')(enc_2)
enc_4 = Dense(125, activation='relu', kernel_initializer='he_uniform
    ')(enc_3)
enc_5 = Dense(32, activation='relu', kernel_initializer='he_uniform'
    )(enc_4)
dec_1 = Dense(125, activation='relu', kernel_initializer='he_uniform
    ')(enc_5)
dec_2 = Dense(250, activation='relu', kernel_initializer='he_uniform
    ')(dec_1)
dec_3 = Dense(500, activation='relu', kernel_initializer='he_uniform
    ')(dec_2)
dec_4 = Dense(1000, activation='relu', kernel_initializer='
    he_uniform')(dec_3)
output_layer = Dense(784, activation='sigmoid')(dec_4)
autoencoder = Model(input_layer, output_layer)
```

A continuación se crea la estructura de la red neuronal, con la disposición de nodos y capas que ya ha sido discutida previamente, al igual que las funciones de activación y las metodologías de inicialización de pesos y *bias*.

```
adam_optim = optimizers.Adam(lr = 0.001)
autoencoder.compile(optimizer=adam_optim, loss='mean_squared_error',
     metrics=['accuracy'])
autoencoder.summary()
```

Posteriormente, se establece el optimizador con la tasa de aprendizaje determinada, así como la metodología de cálculo de pérdidas que se va a utilizar, en este caso el error cuadrático medio entre la entrada y la salida de la red.

```
(x_train, _), (x_test, _) = fashion_mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:]))
    )
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
autoencoder.fit(x_train,x_train,epochs=1000,batch_size=500,
    validation_data=(x_test, x_test))
```

Por último, se cargan los conjuntos de datos de entrenamiento y prueba, y a continuación se normalizan y se reorganizan para tener una única dimensión. Finalmente se emplea la función de *Keras fit* para el entrenamiento del *autoencoder*, utilizando como argumentos la misma base de datos para la entrada y el objetivo con el que debe aprender, el número de épocas y el tamaño del bloque de información.

## 4.5. Elephas

A continuación se presentan las líneas de código que se añaden a *Keras* para realizar la ejecución distribuida mediante la extensión *Elephas*:

```
from pyspark import SparkContext, SparkConf
MASTER="mesos://zk://10.0.12.77:2181,10.0.12.78:2181,
    10.0.12.51:2181,10.0.12.60:2181,10.0.12.75:2181,
    10.0.12.76:2181,10.0.12.18:2181/mesos"
conf = (SparkConf().setAppName("AeSparkKerasTF").
       setMaster(MASTER).
       set("spark.pyspark.python","/usr/bin/python3").
       set("spark.pyspark.driver.python","/usr/bin/python3").
       set("spark.executor.cores","4").
       set("spark.cores.max","20").
       set("spark.executor.memory","16G").
       set("spark.executorEnv.KERAS_BACKEND","tensorflow").
       set("spark.rpc.message.maxSize","200")
     )
sc = SparkContext(conf=conf)
```

Mediante este extracto se configura la aplicación de *Spark*, a la que se indican multitud de parámetros, siendo los más relevantes en este caso el *backend* a utilizar, el número de núcleos o *cores* de que dispondrá cada máquina, el total de los mismos y la cantidad de memoria que será asignada a cada uno de esos trabajadores o máquinas, cuyo número será de $4G \times n_{nucleos}$. Al código anterior, en el caso de *Theano*, se añadirá o modificará lo siguiente en la inicialización de la variable *conf*:

```
set("spark.executorEnv.KERAS_BACKEND","theano").
set("spark.executorEnv.THEANO_FLAGS",'device=cpu, floatX=float32,
    base_compiledir=/var/tmp, mode=fast_run, image_data_format=
    channels_first').
```

Por último, al final del código se añaden las siguientes líneas para ejecutarlo de forma distribuida mediante *Elephas*:

```
from elephas.utils.rdd_utils import to_simple_rdd
from elephas.spark_model import SparkModel
rdd = to_simple_rdd(sc, x_train, x_train)
spark_model = SparkModel(autoencoder, frequency='epoch', mode='
    synchronous', batch_size=500, num_workers=5)
spark_model.fit(rdd, epochs=1000, batch_size=500, verbose=1)
```

Sobre esto, se debe concretar que *Spark* hace uso de una variable abstracta llamada RDD (*Resilient Distributed Dataset*, conjunto de datos distribuidos resiliente), que supone un conjunto de datos o elementos, particionados en diferentes nodos del clúster con el que se esté trabajando, sobre el que se puede realizar operaciones en paralelo [84]. En el caso tratado, las bases de datos se convierten a RDD, y posteriormente se alimentan al modelo de *Spark*, que a continuación es entrenado de forma similar a como se hacía en *Keras*, aunque ahora se añade además el número de trabajadores o máquinas que se van a utilizar.

CAPÍTULO 5

# EXPERIMENTOS Y ANÁLISIS DE RESULTADOS

## 5.1. Entorno de simulación

En esta sección se detallarán las características del entorno en el que se han realizado las ejecuciones, que consiste principalmente en dos granjas de computación, la primera de las cuales cuenta con 8 GPU independientes, además de multitud de CPU, que se pueden asignar a los procesos que se vayan a ejecutar. En este medio se realizaron la mayor parte de los experimentos, concretamente los que no tuvieron que ver con *Elephas* y *Spark*, debido a que no se pudieron efectuar pruebas con estas librerías y extensiones en un entorno distribuido con GPU, pues la paralelización de los procesos, es decir, su distribución entre los distintos núcleos, no funcionaba correctamente y a menudo desembocaba en errores. Es por esto que se recurrió a la segunda granja de simulación, que no cuenta con GPU, pero sí con una gran cantidad de CPU, ya que se encuentra más orientada hacia la implementación de aplicaciones para procesado *Big Data*. Ambos medios hacen uso de *Apache Mesos*, un administrador de recursos para entornos virtuales, centros de datos y otros sistemas que cuenten con grandes cantidades de información, a los cuales organiza para realizar ejecuciones distribuidas con una alta escalabilidad y tolerancia ante errores [85].

## 5.2. Comentario previo a los experimentos

A modo de resumen de los principales experimentos realizados se presenta la tabla 5.1, que detalla los diferentes cambios que se han ido efectuando con el objetivo de reducir el error de validación obtenido al entrenar el *autoencoder*.

| Experimento | Tamaño del bloque | Épocas | Inicialización pesos | Inicialización *bias* | Tasa aprendizaje | Función activación |
|---|---|---|---|---|---|---|
| I | 240 | 1000 | Aleatoria uniforme | Aleatoria uniforme | 0.1 | Sigmoide |
| II | 240 | 1000 | Aleatoria uniforme | Aleatoria uniforme | 0.01 | Sigmoide |
| III | 240 | 1000 | Aleatoria uniforme | Aleatoria uniforme | 0.001 | Sigmoide |
| IV | 240 | 1000 | Aleatoria normal | Aleatoria normal | 0.001 | Sigmoide |
| V | 500 | 1000 | Aleatoria normal | Aleatoria normal | 0.001 | Sigmoide |
| VI | 500 | 1000 | Aleatoria normal | Aleatoria normal | 0.001 | ReLU + sigmoide |
| VII | 500 | 1000 | Xavier normal | Xavier normal | 0.001 | ReLU + sigmoide |
| VIII | 500 | 1000 | Xavier normal | Xavier normal | 0.0001 | ReLU + sigmoide |

Tabla 5.1. Resumen detallado de los principales experimentos realizados.

De la tabla anterior (5.1) se debe destacar que los parámetros que no aparecen en ella, como el número de capas o los nodos contenidos en cada una de ellas, no varían en ningún momento y son los mismos que se indicaron en el capítulo anterior. En las secciones posteriores de este capítulo se ofrecerá una breve descripción de los cambios realizados en cada experimento, así como gráficas comparativas entre las distintas librerías que ilustren los avances que se han conseguido con ellos. Debido al carácter aleatorio de la inicialización de varios parámetros, dichas figuras se han obtenido tras haber realizado, para cada caso, cinco simulaciones, de las cuales se ha obtenido un valor medio y su correspondiente varianza, que han sido empleadas para la representación.

## 5.3. Experimento I

La siguiente tabla (5.2) se presenta como un resumen por capas de los parámetros que se utilizarán en este experimento, los cuales se detallarán después.

| Nº nodos | Inicialización de pesos | Inicialización de *bias* | Función de activación |
|---|---|---|---|
| 784 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 1000 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 500 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 250 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 125 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 32 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 125 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 250 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 500 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 1000 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 784 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |

Tabla 5.2. Detalle del experimento I

- Tasa de aprendizaje: se ha comenzado eligiendo un decimal elevado, 0.1, con el objeto de ir disminuyéndolo en los sucesivos experimentos.
- Tamaño del bloque (*batch*): se ha escogido el valor de 240 para que el conjunto de entrenamiento se agrupe en $60000/240 = 250$ divisiones, que serán empleadas para entrenar la red.
- Número de épocas: la cifra de 1000 se seleccionó de forma arbitraria para que no sea muy pequeña, con lo que no se podría ver correctamente el error alcanzado, al menos hasta un cierto umbral; ni muy alta, lo cual podría alargar demasiado el tiempo de ejecución.
- Inicialización de pesos: se ha escogido una distribución aleatoria uniforme, entre 0 y 1, para comenzar debido a su simpleza.
- Inicialización de *bias*: al igual que en el anterior caso se ha escogido una distribución aleatoria uniforme, con las mismas condiciones.

- Función de activación: se ha seleccionado la función sigmoide en la última capa ya que por su funcionamiento retorna valores entre 0 y 1. En el resto de la red se ha seleccionado la misma con el objetivo de homogeneizarla.
- Optimizador: se emplea el algoritmo Adam, cuyo funcionamiento se detallará en el análisis completo.

A continuación se muestran las gráficas obtenidas al aplicar los parámetros antes descritos a un *autoencoder*, utilizando los set de datos de *MNIST* y *Fashion-MNIST*.

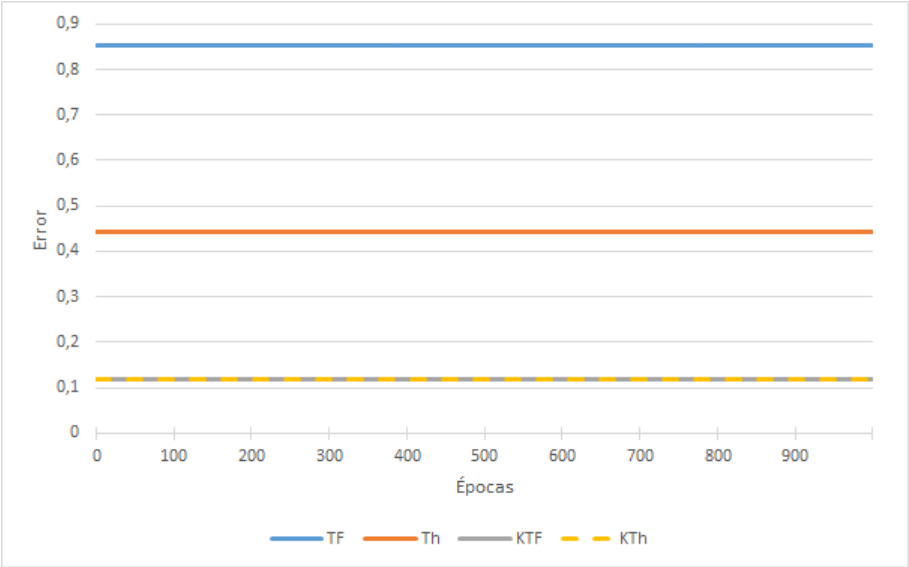



Fig. 5.1. Resultados del experimento I en *MNIST*

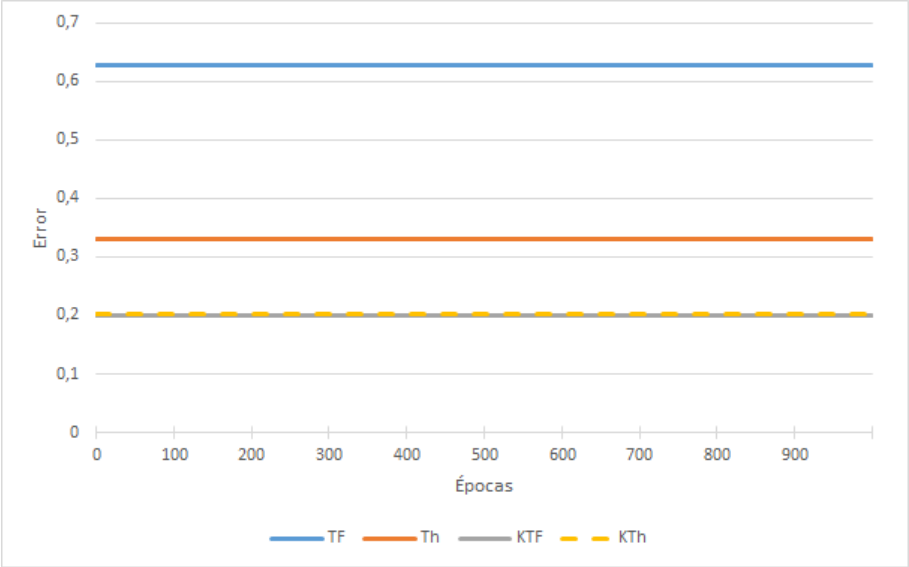



Fig. 5.2. Resultados del experimento I en *Fashion-MNIST*

Como se observa, el error en los 8 casos presentados en las gráficas no varía, debido principalmente a la alta tasa de aprendizaje que fue escogida, lo cual ya fue previsto y planteado antes de la realización del experimento.

## 5.4. Experimento II

De forma similar al anterior caso, se presenta a continuación una tabla (5.3) a modo de resumen por capas de los parámetros que se utilizarán en este experimento, los cuales serán detallados después.

| Nº nodos | Inicialización de pesos | Inicialización de *bias* | Función de activación |
|---|---|---|---|
| 784 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 1000 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 500 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 250 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 125 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 32 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 125 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 250 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 500 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 1000 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 784 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |

Tabla 5.3. Detalle del experimento II

- Tasa de aprendizaje: para paliar el principal problema del experimento anterior, se ha disminuido en un orden de magnitud el valor seleccionado, que ahora es 0.01.
- Tamaño del bloque (*batch*): se mantiene el anterior, 240.
- Número de épocas: sigue siendo el mismo, 1000.
- Inicialización de pesos: se conserva la distribución aleatoria uniforme, entre 0 y 1.
- Inicialización de *bias*: al igual que en el anterior caso se mantiene la distribución aleatoria uniforme, con las mismas condiciones.
- Función de activación: sigue siendo la función sigmoide para toda la red.
- Optimizador: se continúa utilizando Adam.

Abajo se muestran las gráficas pertenecientes a la ejecución del modelo sobre las bases de datos de *MNIST* y *Fashion-MNIST*.

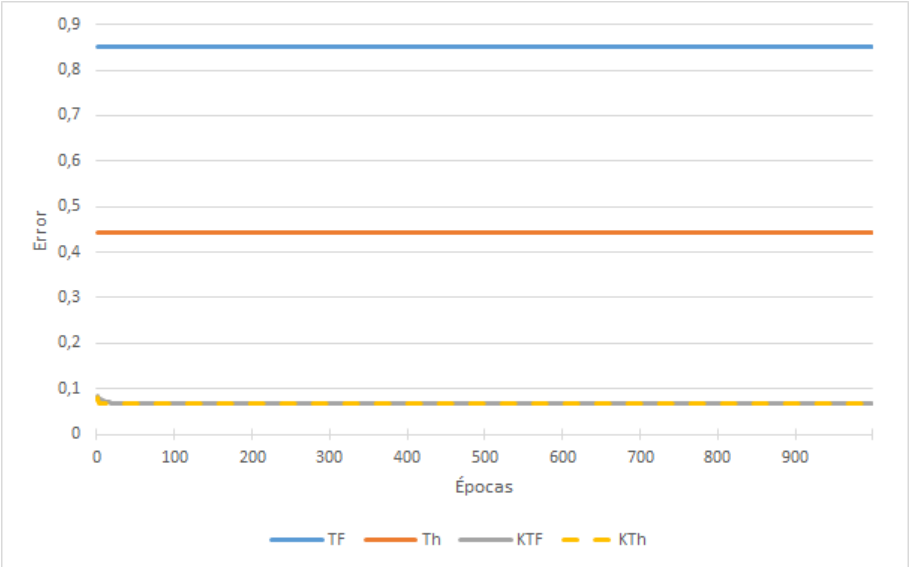



Fig. 5.3. Resultados del experimento II en *MNIST*

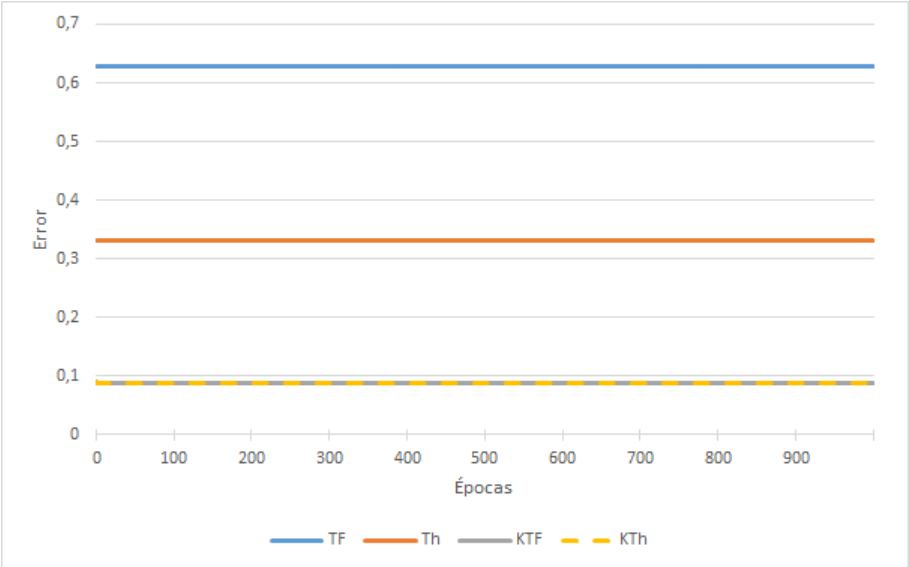



Fig. 5.4. Resultados del experimento II en *Fashion-MNIST*

Como se aprecia al comparar este caso con el primero, los resultados tanto de *TensorFlow*, como de *Theano*, no varían con la disminución en la tasa de aprendizaje. No es así con *Keras*, donde sí que son mejores, e incluso se aprecia una ligera bajada en las primeras épocas, más clara en la gráfica perteneciente a *MNIST*.

## 5.5. Experimento III

Para esta prueba se plantea la misma estructura que en las anteriores, en primer lugar aparece una tabla (5.4) que resume por capas los parámetros que se utilizarán ahora, los cuales se detallarán después.

| Nº nodos | Inicialización de pesos | Inicialización de *bias* | Función de activación |
|---|---|---|---|
| 784 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 1000 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 500 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 250 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 125 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 32 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 125 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 250 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 500 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 1000 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |
| 784 | Aleatoria uniforme | Aleatoria uniforme | Sigmoide |

Tabla 5.4. Detalle del experimento III

- Tasa de aprendizaje: se ha actuado de igual manera que en el experimento anterior, disminuyendo en un orden de magnitud el valor, por lo que ahora es 0.001.
- Tamaño del bloque (*batch*): se conserva el anterior, 240.
- Número de épocas: continúa siendo el mismo, 1000.
- Inicialización de pesos: se mantiene la distribución aleatoria uniforme, entre 0 y 1.
- Inicialización de *bias*: es la misma metodología que para los pesos, aleatoria uniforme, con las mismas condiciones.
- Función de activación: se utiliza la función sigmoide para toda la red.
- Optimizador: se mantiene el algoritmo Adam, empleado en los experimentos anteriores.

Se presentan a continuación los resultados obtenidos al aplicar los parámetros anteriormente definidos en un *autoencoder* con los conjuntos de datos de *MNIST* y *Fashion-MNIST*.

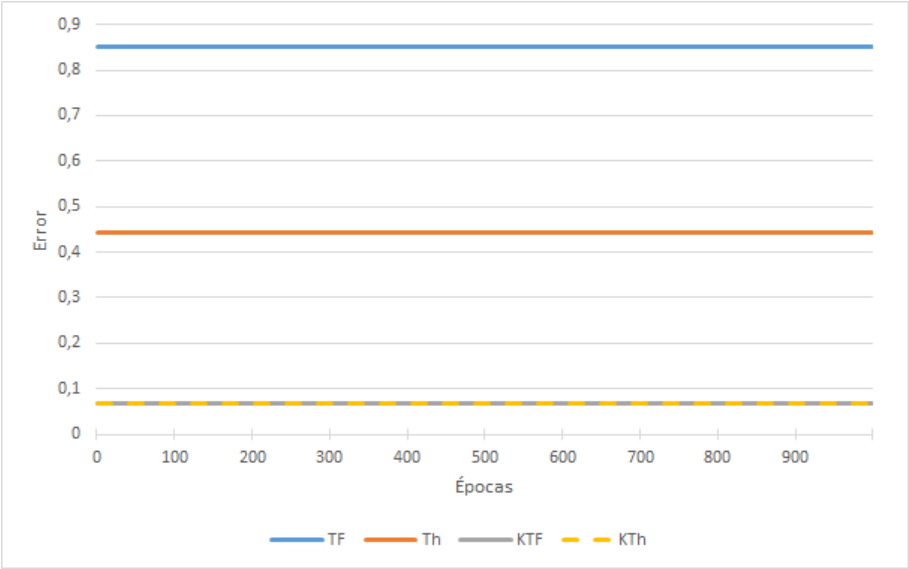



Fig. 5.5. Resultados del experimento III en *MNIST*

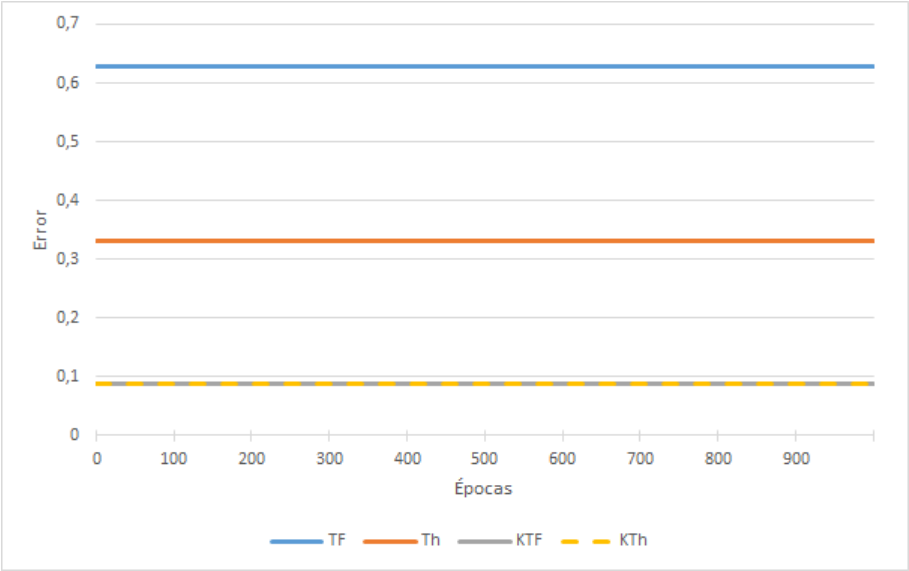



Fig. 5.6. Resultados del experimento III en *Fashion-MNIST*

En comparación con el experimento anterior, los resultados no han variado, lo cual podría ser indicativo de que haya que modificar otros parámetros, como la inicialización de pesos y *bias*, o su optimización.

## 5.6. Experimento IV

De la misma manera que en los anteriores casos, a continuación se presenta una tabla (5.5), en la que se indican, separados por capas, los parámetros que se utilizarán en este experimento. La elección de dichos valores se especificará posteriormente.

| Nº nodos | Inicialización de pesos | Inicialización de *bias* | Función de activación |
|---|---|---|---|
| 784 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 1000 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 500 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 250 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 125 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 32 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 125 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 250 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 500 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 1000 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 784 | Aleatoria normal | Aleatoria normal | Sigmoide |

Tabla 5.5. Detalle del experimento IV

- Tasa de aprendizaje: se utiliza el valor del experimento previo, 0.001.
- Tamaño del bloque (*batch*): se sigue manteniendo el anterior, 240.
- Número de épocas: continúa siendo el mismo, 1000.
- Inicialización de pesos: se ha cambiado la distribución aleatoria uniforme a normal, centrada en 0 y con una varianza de 1, ambos siendo los valores por defecto que ofrecen las funciones empleadas para la inicialización.
- Inicialización de *bias*: al igual que en el anterior caso se cambia la distribución aleatoria uniforme a normal, con las mismas condiciones.
- Función de activación: sigue siendo la función sigmoide en todo el modelo.
- Optimizador: se emplea el algoritmo Adam, al igual que en los experimentos previos.

Se presentan a continuación los resultados obtenidos al aplicar los parámetros anteriormente definidos en un *autoencoder* a los conjuntos de datos de *MNIST* y *Fashion-MNIST*.

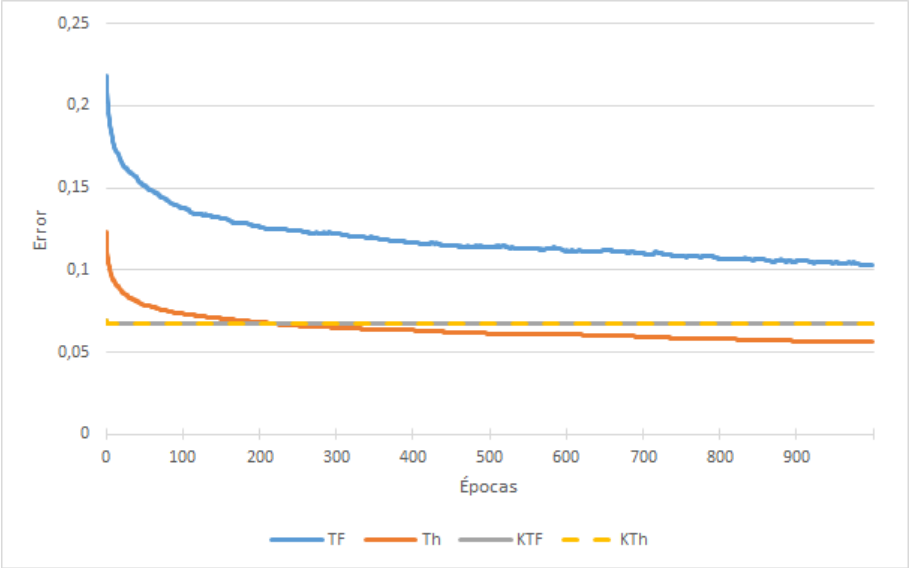



Fig. 5.7. Resultados del experimento IV en *MNIST*

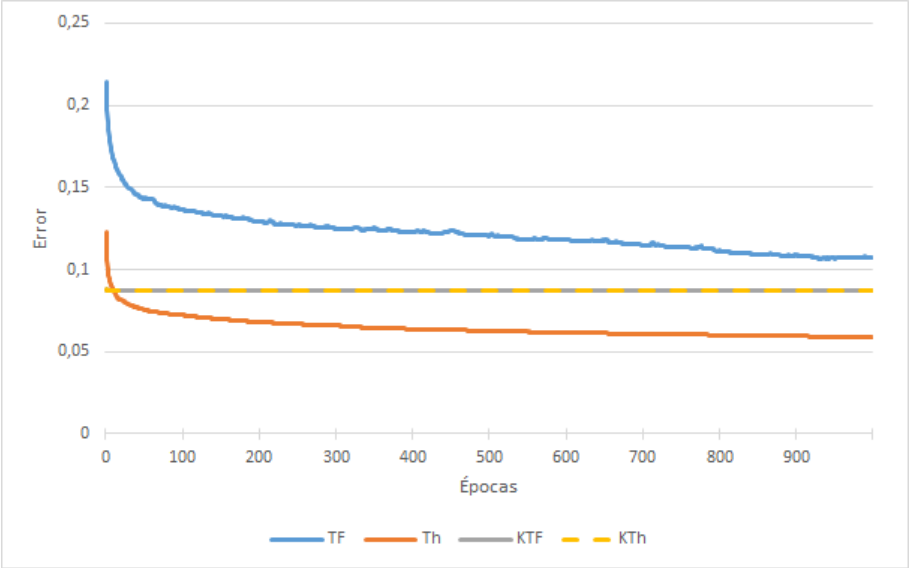



Fig. 5.8. Resultados del experimento IV en *Fashion-MNIST*

En las figuras superiores se observa cómo, gracias al cambio en la inicialización de los pesos y *bias*, se ha conseguido finalmente que el error disminuya, al menos en los casos de *TensorFlow* y *Theano*, que sí describen una curva descendente y que llega a estabilizarse alrededor de la época 900. En *Keras* sigue produciéndose el mismo problema que antes, por lo que habrá que realizar más modificaciones en los parámetros.

## 5.7. Experimento V

Como en los anteriores casos, se presenta a continuación una tabla (5.6) a modo de resumen por capas de los parámetros que se utilizarán en este experimento, los cuales serán detallados después.

| Nº nodos | Inicialización de pesos | Inicialización de *bias* | Función de activación |
|---|---|---|---|
| 784 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 1000 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 500 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 250 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 125 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 32 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 125 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 250 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 500 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 1000 | Aleatoria normal | Aleatoria normal | Sigmoide |
| 784 | Aleatoria normal | Aleatoria normal | Sigmoide |

Tabla 5.6. Detalle del experimento V

- Tasa de aprendizaje: se ha conservado el valor del experimento anterior, 0.001.
- Tamaño del bloque (*batch*): se ha aumentado a 500, de tal forma que el conjunto de entrenamiento se agrupa ahora en $60000/500 = 120$ divisiones. Este aumento se debe a las recomendaciones que se suelen hacer [86] para aumentar el tamaño del bloque en casos como el del optimizador Adam, consiguiendo un efecto en la reducción del error similar al producido al disminuir la tasa de aprendizaje.
- Número de épocas: sigue siendo el mismo, 1000.
- Inicialización de pesos: se mantiene la distribución aleatoria normal, centrada en 0 y con varianza 1.

- Inicialización de *bias*: se conserva también la misma distribución aleatoria normal.
- Función de activación: se continúa usando la función sigmoide para todas las capas.
- Optimizador: se conserva Adam, empleado en los experimentos previos.

Abajo se muestran las gráficas pertenecientes a la ejecución del modelo sobre las bases de datos de *MNIST* y *Fashion-MNIST*.

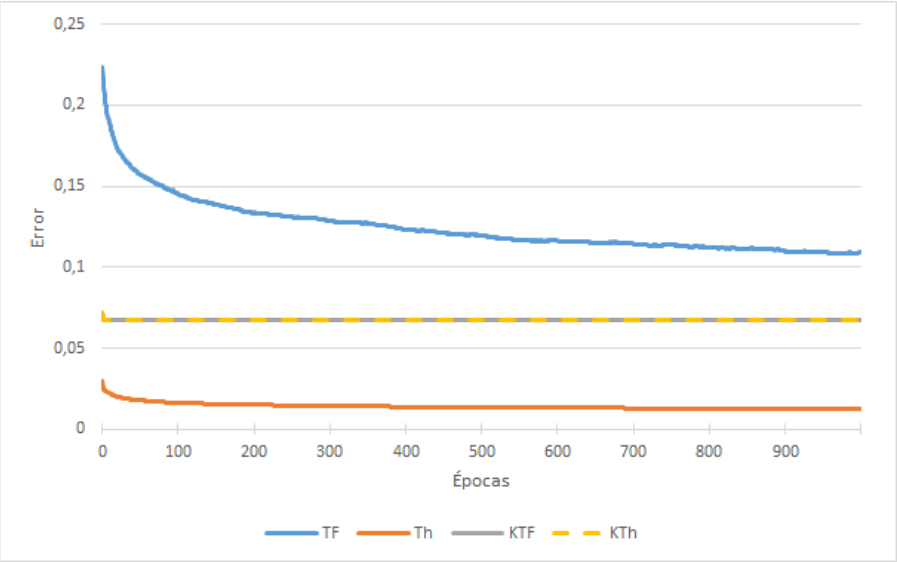



Fig. 5.9. Resultados del experimento V en *MNIST*

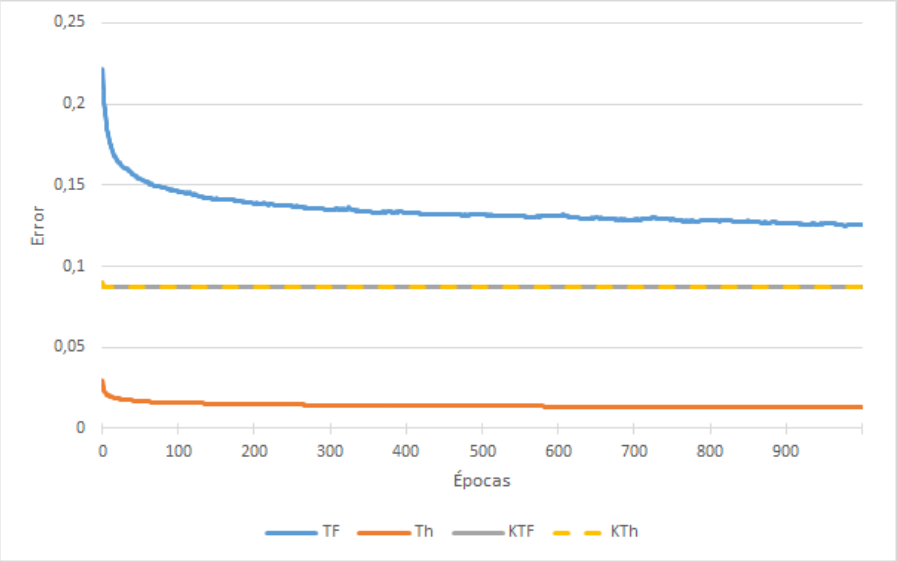



Fig. 5.10. Resultados del experimento V en *Fashion-MNIST*

En comparación con el anterior experimento, en el caso de *Theano* se varían los resultados, de forma satisfactoria además; mientras que en *TensorFlow* lo hacen de manera

ligeramente negativa. En cuanto a *Keras*, es probable que la razón por la que sus valores de error no hayan cambiado de manera significativa se deba a que la función sigmoide esté saturada, bien por que los pesos son demasiado pequeños, o grandes [87].

## 5.8. Experimento VI

Para esta prueba se plantea la misma estructura que en las anteriores, en primer lugar aparece una tabla (5.7) que resume por capas los parámetros que se utilizarán ahora, los cuales se detallarán después.

| Nº nodos | Inicialización de pesos | Inicialización de *bias* | Función de activación |
|---|---|---|---|
| 784 | Aleatoria normal | Aleatoria normal | ReLU |
| 1000 | Aleatoria normal | Aleatoria normal | ReLU |
| 500 | Aleatoria normal | Aleatoria normal | ReLU |
| 250 | Aleatoria normal | Aleatoria normal | ReLU |
| 125 | Aleatoria normal | Aleatoria normal | ReLU |
| 32 | Aleatoria normal | Aleatoria normal | ReLU |
| 125 | Aleatoria normal | Aleatoria normal | ReLU |
| 250 | Aleatoria normal | Aleatoria normal | ReLU |
| 500 | Aleatoria normal | Aleatoria normal | ReLU |
| 1000 | Aleatoria normal | Aleatoria normal | ReLU |
| 784 | Aleatoria normal | Aleatoria normal | Sigmoide |

Tabla 5.7. Detalle del experimento VI

- Tasa de aprendizaje: se mantiene el valor anterior, 0.001.
- Tamaño del bloque (*batch*): se ha conservado el del experimento previo, 500.
- Número de épocas: continúa siendo el mismo, 1000.
- Inicialización de pesos: se mantiene la distribución aleatoria normal, centrada en 0 y con varianza 1.
- Inicialización de *bias*: al igual que en el anterior caso se utiliza la distribución normal, con los mismos parámetros.
- Función de activación: se ha cambiado para todas las capas menos la última, por motivos detallados anteriormente, a la función ReLU, que posee una mayor sencillez y un coste computacional más bajo que la función sigmoide.

- Optimizador: se emplea Adam, como en los experimentos anteriores.

Se presentan a continuación los resultados obtenidos al aplicar los parámetros anteriormente definidos a un *autoencoder* con los conjuntos de datos de *MNIST* y *Fashion-MNIST*.

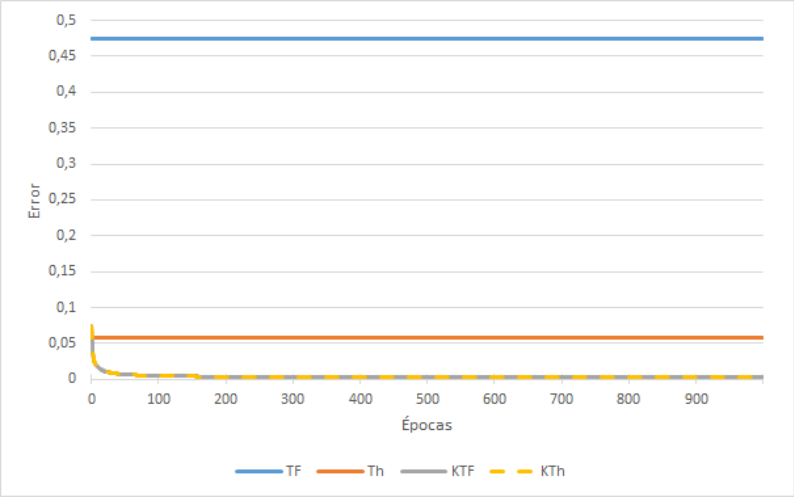



Fig. 5.11. Resultados del experimento VI en *MNIST*

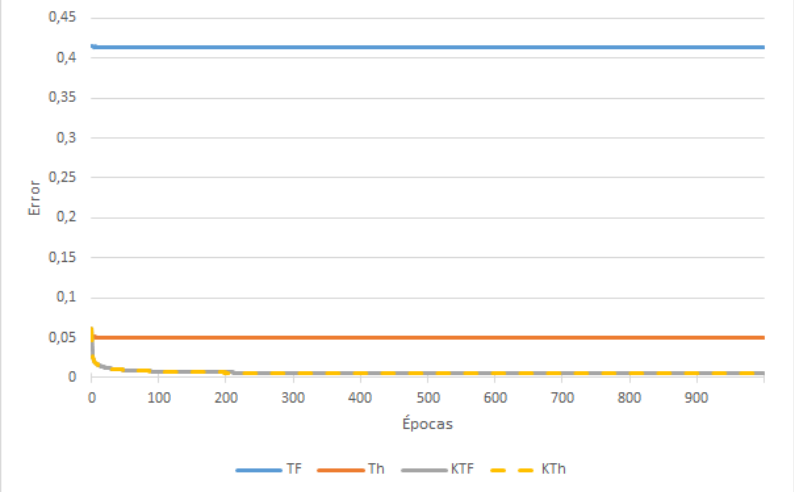



Fig. 5.12. Resultados del experimento VI en *Fashion-MNIST*

Como se ha podido observar, al cambiar la función de activación de la práctica totalidad de la red, se ha producido un fenómeno similar al del anterior experimento. Esto tiene una posible explicación, similar a la ofrecida en el caso previo pero para *ReLU*, conocida como *dying ReLU*, que se describirá en la sección de análisis de resultados; y cuya solución pasa por la dada para la función sigmoide, es decir, modificar la inicialización de los parámetros.

## 5.9. Experimento VII

Al igual que en los casos anteriores se indican a continuación (5.8) los parámetros que se utilizarán en este experimento, que serán descritos posteriormente.

| Nº nodos | Inicialización de pesos | Inicialización de *bias* | Función de activación |
|---|---|---|---|
| 784 | Xavier normal | Xavier normal | ReLU |
| 1000 | Xavier normal | Xavier normal | ReLU |
| 500 | Xavier normal | Xavier normal | ReLU |
| 250 | Xavier normal | Xavier normal | ReLU |
| 125 | Xavier normal | Xavier normal | ReLU |
| 32 | Xavier normal | Xavier normal | ReLU |
| 125 | Xavier normal | Xavier normal | ReLU |
| 250 | Xavier normal | Xavier normal | ReLU |
| 500 | Xavier normal | Xavier normal | ReLU |
| 1000 | Xavier normal | Xavier normal | ReLU |
| 784 | Xavier normal | Xavier normal | Sigmoide |

Tabla 5.8. Detalle del experimento VII

- Tasa de aprendizaje: se ha conservado el valor del experimento anterior, 0.001.
- Tamaño del bloque (*batch*): se ha mantenido el previo, 500.
- Número de épocas: sigue siendo el mismo, 1000.
- Inicialización de pesos: se ha cambiado por la metodología de Xavier o Glorot, que consiste en una distribución normal de media 0 y varianza $\sqrt{\frac{2}{n_{in} + n_{out}}}$, donde $n_{in}$ es el número de entradas de la capa, y $n_{out}$, el de salidas [88].
- Inicialización de *bias*: también se ha cambiado al algoritmo de Xavier, con las mismas características.
- Función de activación: se mantiene la estructura del experimento previo, todas las capas ReLU excepto la última, que utiliza la función sigmoide.
- Optimizador: se utiliza Adam, al igual que en los casos anteriores.

Se presentan a continuación los resultados obtenidos al aplicar los parámetros anteriormente definidos en un *autoencoder* con los conjuntos de datos de *MNIST* y *Fashion-MNIST*.

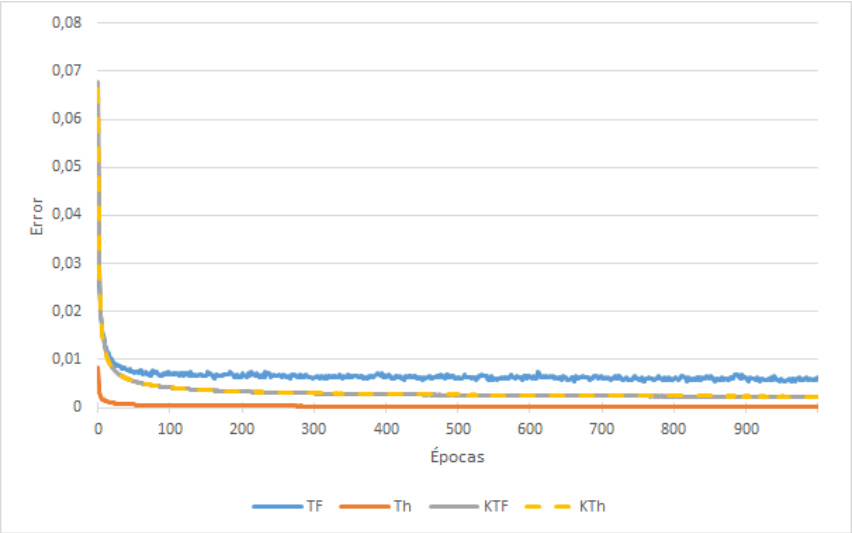



Fig. 5.13. Resultados del experimento VII en *MNIST*

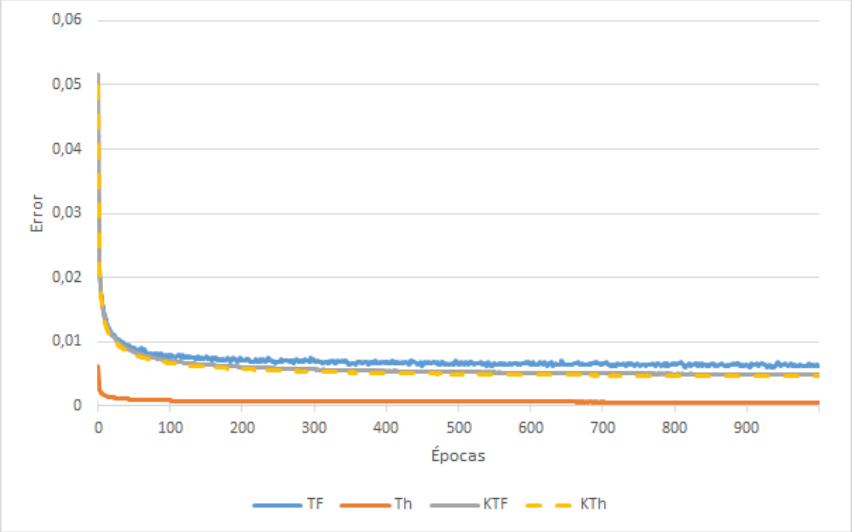



Fig. 5.14. Resultados del experimento VII en *Fashion-MNIST*

Como se desprende de la comparación entre las gráficas de este experimento con el anterior, la modificación en la inicialización de los pesos ha resultado ser, a priori, muy beneficiosa. En los casos de *Theano* y *TensorFlow* se ha conseguido superar el problema planteado anteriormente, además de mejorar los resultados del experimento previo a que se produjera la saturación en las funciones de activación (figuras correspondientes al VI, 5.11 y 5.12). No obstante, se debe tener en cuenta el moderado carácter oscilante que presentan los resultados de *TensorFlow*, los cuales se deberán intentar corregir mediante el uso de otra metodología de inicialización de parámetros, o una menor tasa de aprendizaje.

## 5.10. Experimento VIII

A continuación se describirá en qué ha consistido el experimento, comenzando por una tabla (5.9), que indica las principales características del mismo, las cuales serán posteriormente razonadas.

| Nº nodos | Inicialización de pesos | Inicialización de *bias* | Función de activación |
|---|---|---|---|
| 784 | Xavier normal | Xavier normal | ReLU |
| 1000 | Xavier normal | Xavier normal | ReLU |
| 500 | Xavier normal | Xavier normal | ReLU |
| 250 | Xavier normal | Xavier normal | ReLU |
| 125 | Xavier normal | Xavier normal | ReLU |
| 32 | Xavier normal | Xavier normal | ReLU |
| 125 | Xavier normal | Xavier normal | ReLU |
| 250 | Xavier normal | Xavier normal | ReLU |
| 500 | Xavier normal | Xavier normal | ReLU |
| 1000 | Xavier normal | Xavier normal | ReLU |
| 784 | Xavier normal | Xavier normal | Sigmoide |

Tabla 5.9. Detalle del experimento VIII

- Tasa de aprendizaje: se ha disminuido en un orden de magnitud para intentar mantener una tendencia en los resultados de *TensorFlow* que oscile menos que en el experimento anterior, por lo que su valor es 0.0001.
- Tamaño del bloque (*batch*): es el mismo que en el experimento anterior, 500.
- Número de épocas: se utiliza un valor de 1000, como en los demás casos.
- Inicialización de pesos: se mantiene la metodología de Xavier, con el fin de intentar corregir el problema del experimento anterior.
- Inicialización de *bias*: de nuevo es la misma que para los pesos.
- Función de activación: se conserva el modelo previamente usado, con todas las capas utilizando ReLU excepto la última, que emplea la función sigmoide.
- Optimizador: se mantiene el uso del algoritmo Adam.

Debajo se presentan los resultados obtenidos al aplicar estos parámetros a un *autoencoder* con las bases de datos *MNIST* y *Fashion-MNIST*.

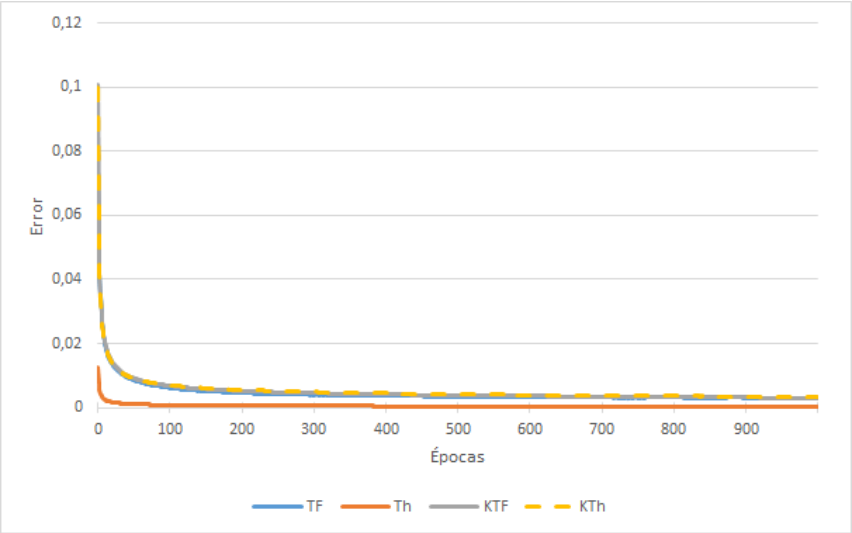



Fig. 5.15. Resultados del experimento VIII en *MNIST*

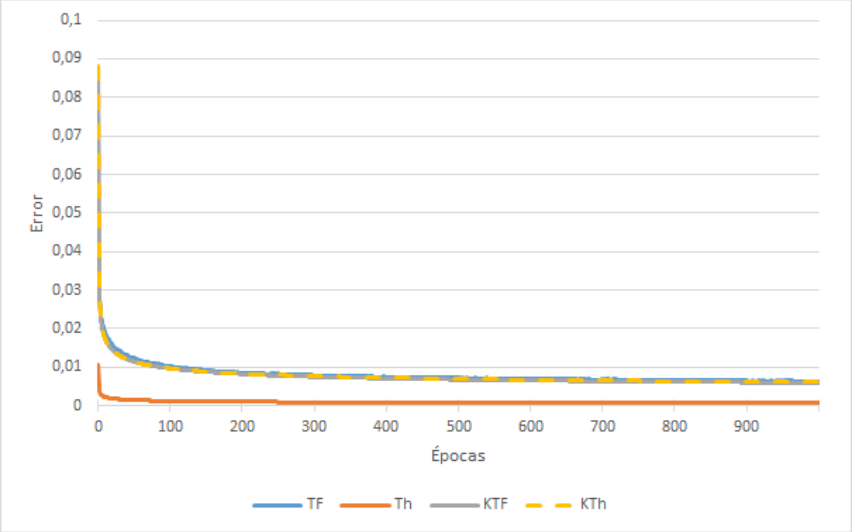



Fig. 5.16. Resultados del experimento VIII en *Fashion-MNIST*

De la observación de las gráficas se puede extraer que únicamente en el caso de *TensorFlow* se ha conseguido mejorar el experimento anterior, además de una tendencia descendente mucho más estable, sin tantas ondulaciones. Para el resto de librerías, no obstante, el cambio ha supuesto un empeoramiento en los resultados obtenidos.

## 5.11. Otros experimentos

A continuación se definen una serie de pruebas que también se realizaron, y que están relacionadas con las principales, descritas antes; pero cuyos resultados, bien no aportan mejoras significativas, o bien son peores. Cada experimento vendrá acompañado de una

tabla, que contendrá las características principales del mismo, a la que seguirá una explicación de por qué no resulta relevante, la cual podrá estar acompañada de gráficos auxiliares que ayuden a comprender la motivación.

### 5.11.1. Experimento III.a

| | |
|---|---|
| Tasa de aprendizaje | 0.0001 |
| Tamaño del bloque (*batch*) | 240 |
| Número de épocas | 1000 |
| Inicialización de pesos | Aleatoria uniforme |
| Inicialización de *bias* | Aleatoria uniforme |
| Función de activación | Sigmoide en todas las capas |
| Optimizador | Adam |

Tabla 5.10. Características principales del experimento III.a

Este experimento fue realizado siguiendo la tendencia de II y III en cuanto a disminuir la tasa de aprendizaje en un orden de magnitud cada vez. Como se puede observar en las gráficas siguientes, los resultados no mejoraron, por tanto, en las pruebas sucesivas se optó por un valor, 0.001, seleccionado por defecto para Adam, el optimizador empleado. En ambas figuras cabe indicar que las curvas de *Keras* (independientemente de su *backend* o de a qué experimento pertenezcan) poseen prácticamente los mismos valores, por lo que aparecen superpuestas y no se aprecian todas ellas.

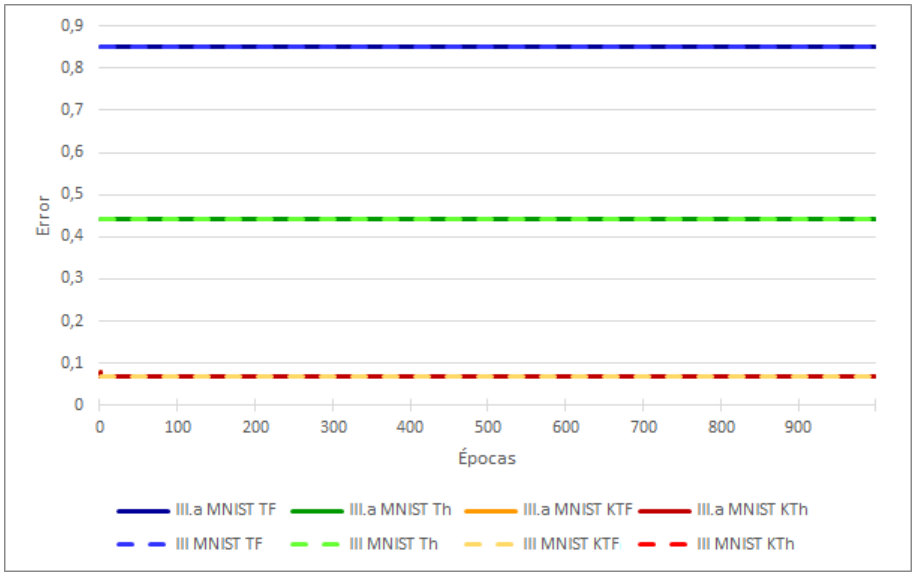



Fig. 5.17. Resultados del experimento III.a en *MNIST*

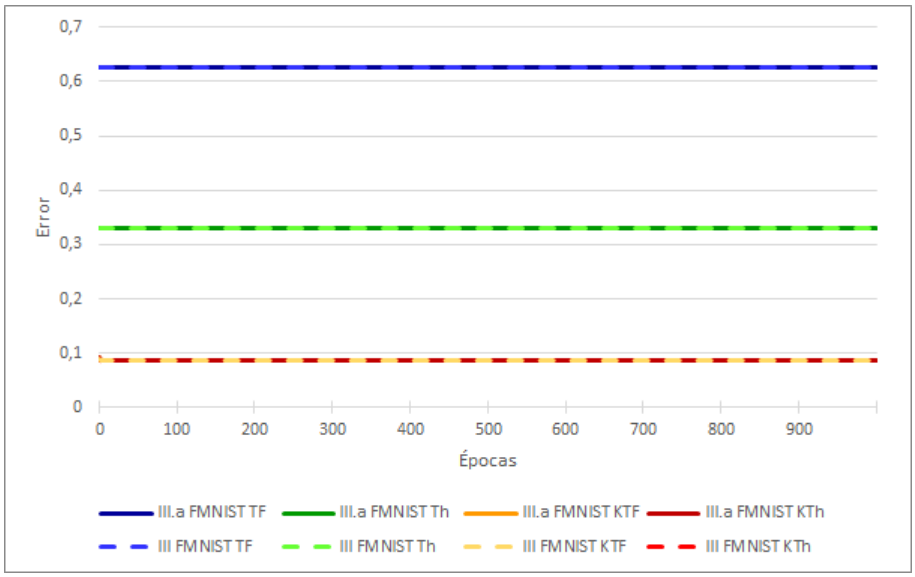



Fig. 5.18. Resultados del experimento III.a en *Fashion-MNIST*

### 5.11.2. Experimento IV.a

| | |
|---|---|
| Tasa de aprendizaje | 0.001 |
| Tamaño del bloque (*batch*) | 240 |
| Número de épocas | 1000 |
| Inicialización de pesos | Aleatoria uniforme |
| Inicialización de *bias* | Aleatoria normal |
| Función de activación | Sigmoide en todas las capas |
| Optimizador | Adam |

Tabla 5.11. Características principales del experimento IV.a

La razón por la que este experimento fue realizado era estudiar cómo afectaba el cambio en la inicialización de los *bias* a los resultados. Como se puede apreciar en las siguientes gráficas, el cambio no supuso una gran mejora, por lo que en pruebas sucesivas fueron principalmente los valores iniciales de los pesos los que se tuvieron en cuenta.

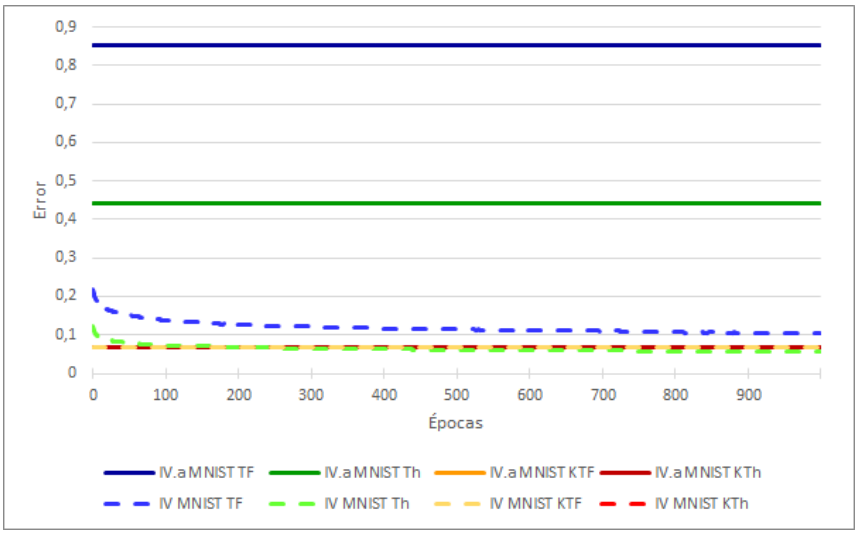



Fig. 5.19. Resultados del experimento IV.a en *MNIST*

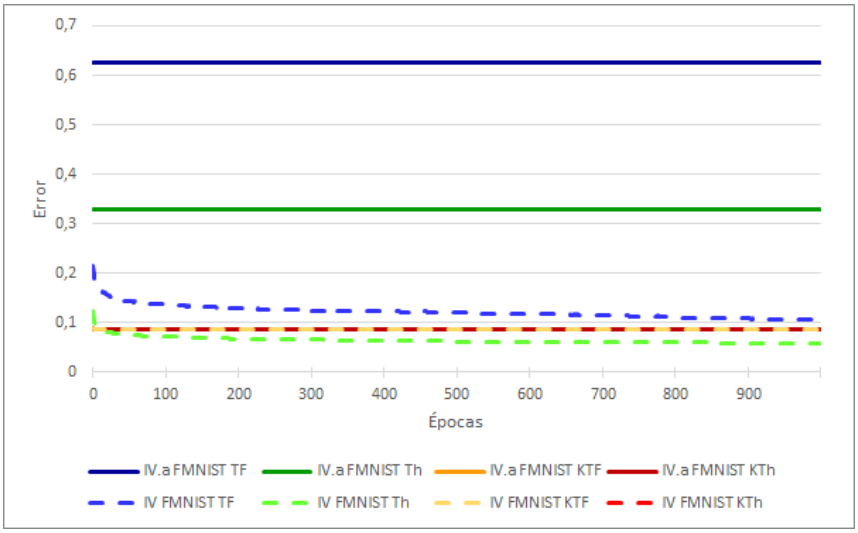



Fig. 5.20. Resultados del experimento IV.a en *Fashion-MNIST*

### 5.11.3. Experimento IV.b

| | |
|---|---|
| Tasa de aprendizaje | 0.001 |
| Tamaño del bloque (*batch*) | 240 |
| Número de épocas | 1000 |
| Inicialización de pesos | Aleatoria normal |
| Inicialización de *bias* | Aleatoria uniforme |
| Función de activación | Sigmoide en todas las capas |
| Optimizador | Adam |

Tabla 5.12. Características principales del experimento IV.b

Exte experimento se llevó a cabo a la vez que el IV.a, con el fin de estudiar la inicialización de qué parámetro, pesos o *bias*, era más definitiva a la hora de mejorar los resultados. Los resultados obtenidos mostraron que se trataba de los primeros, como se puede apreciar al observar las siguientes gráficas, aunque la mejor solución pasaba por utilizar la misma metodología para ambos parámetros, especialmente en el caso de *TensorFlow*.

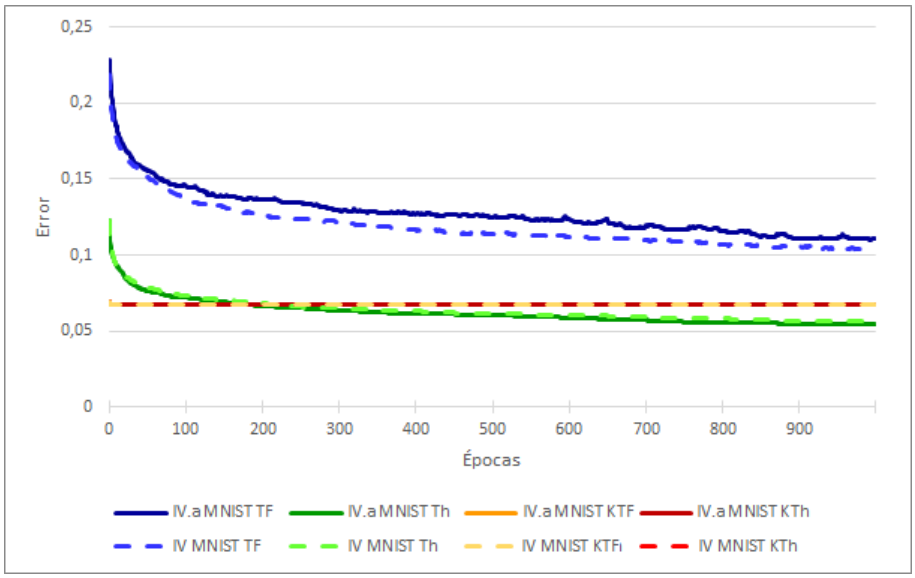



Fig. 5.21. Resultados del experimento IV.b en *MNIST*

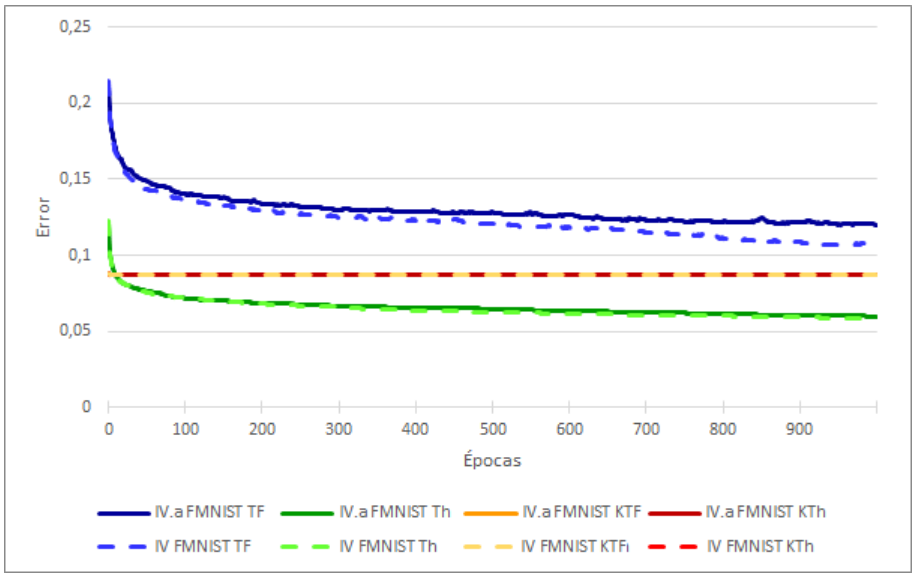



Fig. 5.22. Resultados del experimento IV.b en *Fashion-MNIST*

### 5.11.4. Experimento V.a

| | |
|---|---|
| Tasa de aprendizaje | 0.001 |
| Tamaño del bloque (*batch*) | 2000 |
| Número de épocas | 1000 |
| Inicialización de pesos | Aleatoria normal |
| Inicialización de *bias* | Aleatoria normal |
| Función de activación | Sigmoide en todas las capas |
| Optimizador | Adam |

Tabla 5.13. Características principales del experimento V.a

El objetivo de este experimento era evaluar la variación de los resultados al aumentar el tamaño del bloque de entrenamiento (*batch*) a un valor mayor a 500. De las siguientes gráficas se desprende que el rendimiento, especialmente en el caso de *TensorFlow*, depende de la base de datos, lo cual se razonará más adelante en el análisis general, al igual que un comentario sobre el impacto que produce la variación en este parámetro en el tiempo de ejecución. Para *Keras*, los resultados no varían (cabe indicar que las curvas, independientemente del *backend* empleado, poseen prácticamente los mismos valores, por lo que aparecen superpuestas), mientras que en *Theano* han empeorado en ambos casos con el cambio.

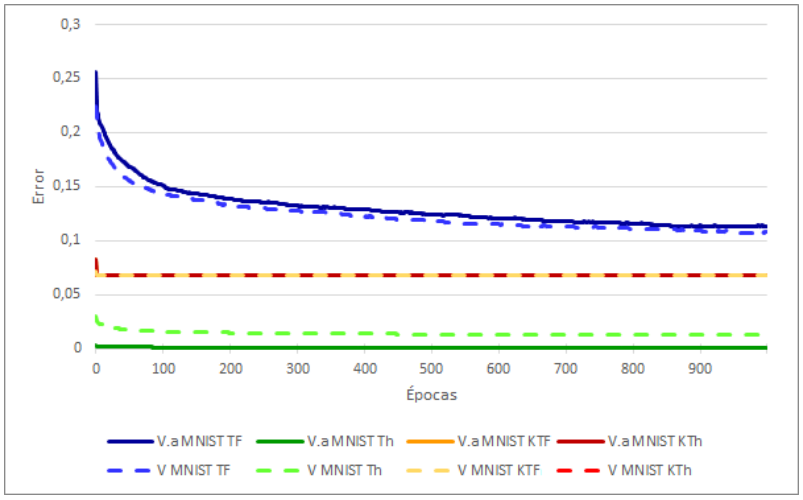



Fig. 5.23. Resultados del experimento V.a en *MNIST*

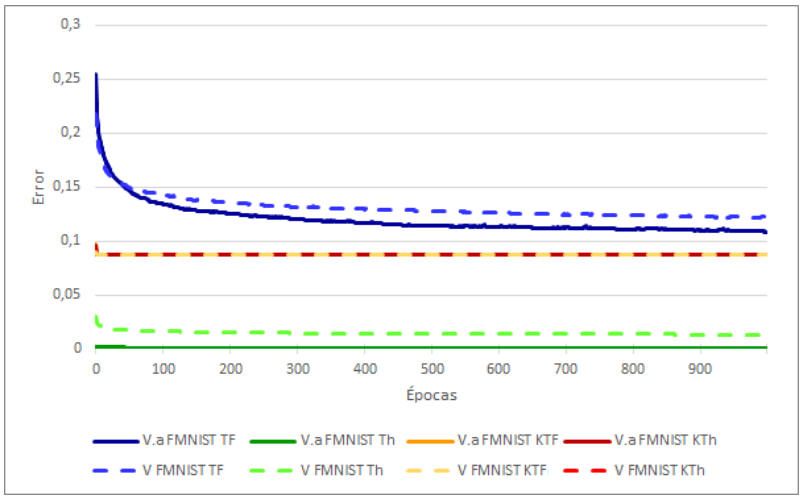



Fig. 5.24. Resultados del experimento V.a en *Fashion-MNIST*

### 5.11.5. Experimento V.b

| | |
|---|---|
| Tasa de aprendizaje | 0.001 |
| Tamaño del bloque (*batch*) | 32 |
| Número de épocas | 1000 |
| Inicialización de pesos | Aleatoria normal |
| Inicialización de *bias* | Aleatoria normal |
| Función de activación | Sigmoide en todas las capas |
| Optimizador | Adam |

Tabla 5.14. Características principales del experimento V.b

Este experimento se planteó como oposición al V.a, con el fin de observar la variación en los resultados obtenidos al utilizar un tamaño menor de 500 para el bloque de entrenamiento. A la luz de las siguientes gráficas se puede concluir que un valor más pequeño produce unas consecuencias sensiblemente negativas en el caso de *Theano*, y algo mejores para *TensorFlow*. De nuevo cabe recalcar que los valores de *Keras* no han variado en función del *backend* utilizado, ni del experimento, por lo que sus curvas se superponen y no es posible apreciarlas todas. Al igual que en el caso previo, merece la pena hacer mención a que el tiempo de ejecución ha sido fuertemente influenciado por el nuevo valor, lo cual se comentará profusamente en el análisis global.

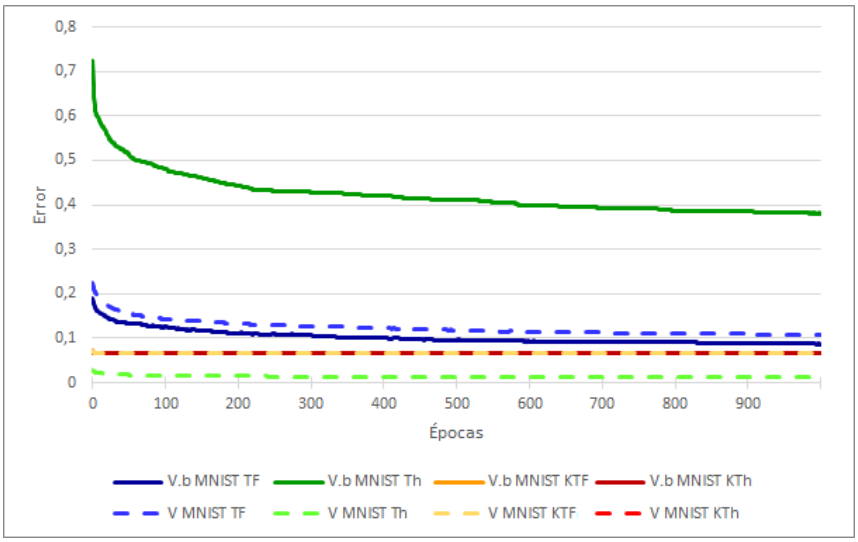



Fig. 5.25. Resultados del experimento V.b en *MNIST*

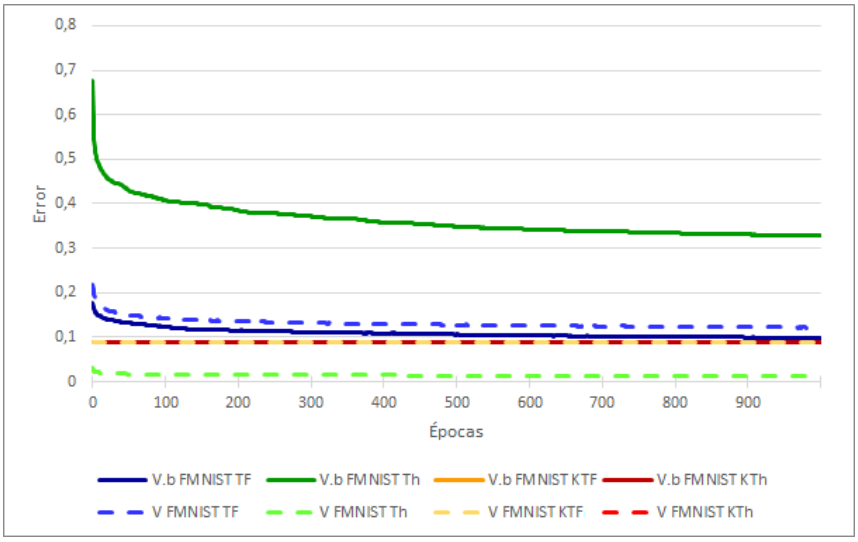



Fig. 5.26. Resultados del experimento V.b en *Fashion-MNIST*

### 5.11.6. Experimento VI.a

| | |
|---|---|
| Tasa de aprendizaje | 0.001 |
| Tamaño del bloque (*batch*) | 500 |
| Número de épocas | 1000 |
| Inicialización de pesos | Aleatoria normal |
| Inicialización de *bias* | Aleatoria normal |
| Función de activación | Sigmoide en todas las capas excepto tangente hiperbólica en la última |
| Optimizador | Adam |

Tabla 5.15. Características principales del experimento VI.a

A pesar de que, a priori, el uso de una función de activación como la tangente hiperbólica, que no retorna valores comprendidos en el rango $[0, 1]$, no es recomendable al sí estarlo los datos alimentados a la entrada de la red, y tratarse de un *autoencoder*; se decidió probar a utilizar dicha función en la última capa del modelo con el fin de averiguar qué ocurría. De este experimento salen las siguientes gráficas, de las cuales se puede inferir que su uso no ha sido beneficioso en ningún caso, de nuevo, probablemente, debido a su forma.

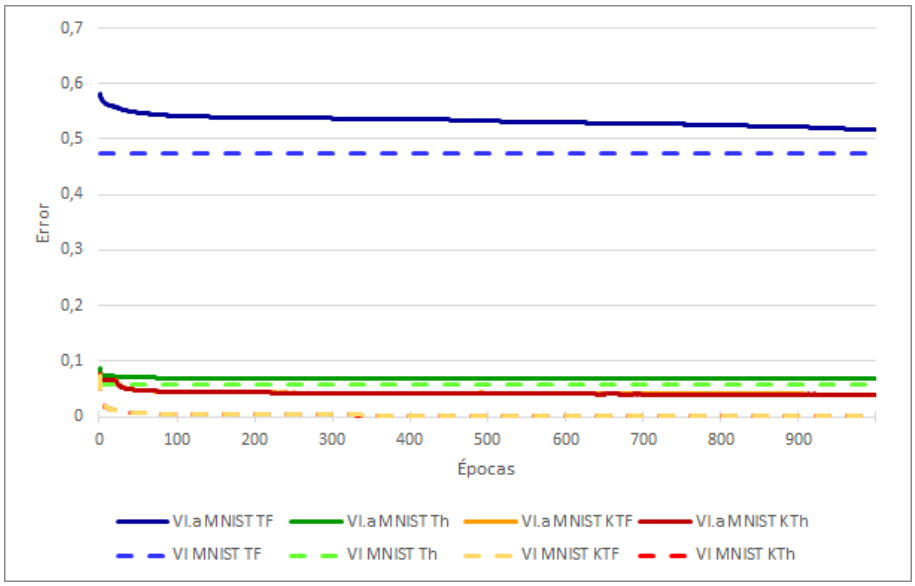



Fig. 5.27. Resultados del experimento VI.a en *MNIST*

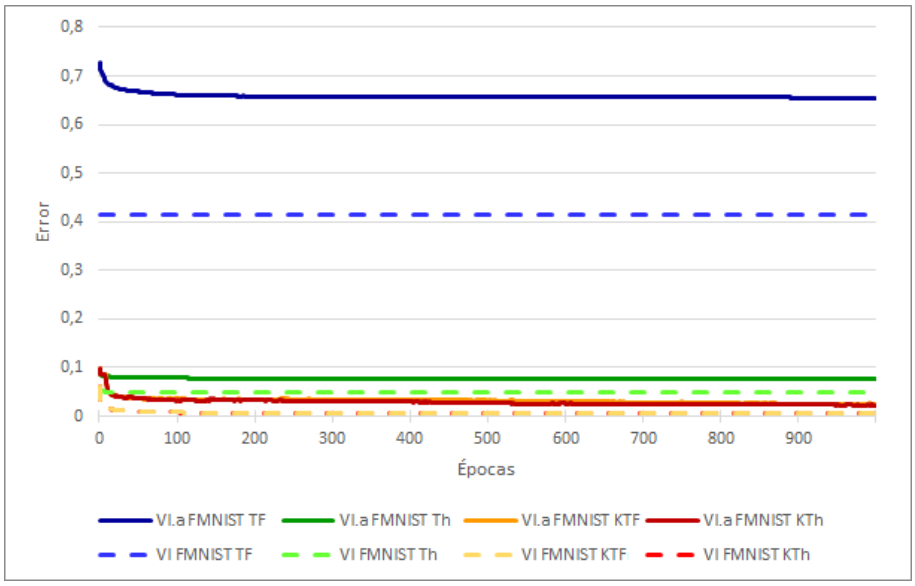



Fig. 5.28. Resultados del experimento VI.a en *Fashion-MNIST*

### 5.11.7. Experimento VII.a

| | |
|---|---|
| Tasa de aprendizaje | 0.001 |
| Tamaño del bloque (*batch*) | 500 |
| Número de épocas | 1000 |
| Inicialización de pesos | Xavier uniforme |
| Inicialización de *bias* | Xavier uniforme |
| Función de activación | ReLU en todas las capas excepto sigmoide en la última |
| Optimizador | Adam |

Tabla 5.16. Características principales del experimento VII.a

A pesar de que, a raíz del experimento IV, se haya utilizado una distribución normal para la inicialización de pesos y *bias*, al haber mejorado sensiblemente los resultados con la metodología de Xavier, se ha decidido probar con su versión uniforme por si estos lo hacían aún más. De las figuras siguientes se concluye que no ha sido así, han sido muy similares en todos los casos, pero ligeramente inferiores en la mayor parte de ellos en las últimas épocas frente al modelo normal.

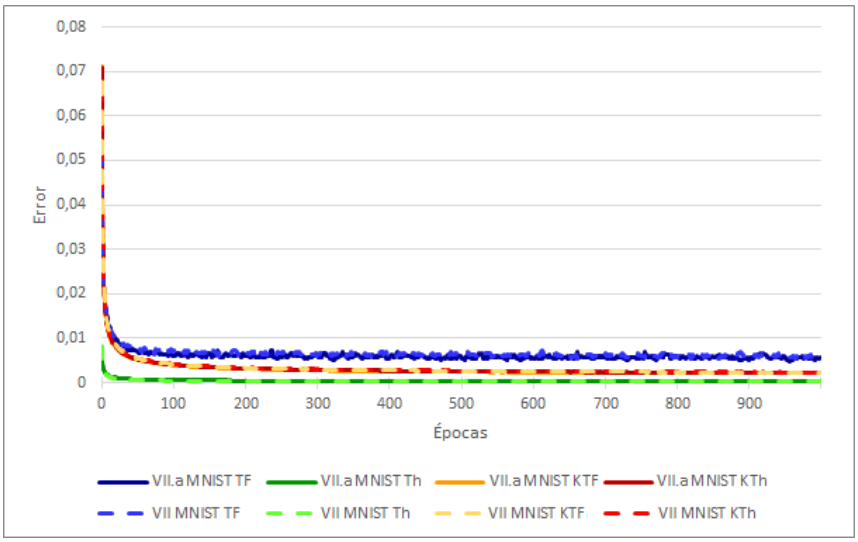



Fig. 5.29. Resultados del experimento VII.a en *MNIST*

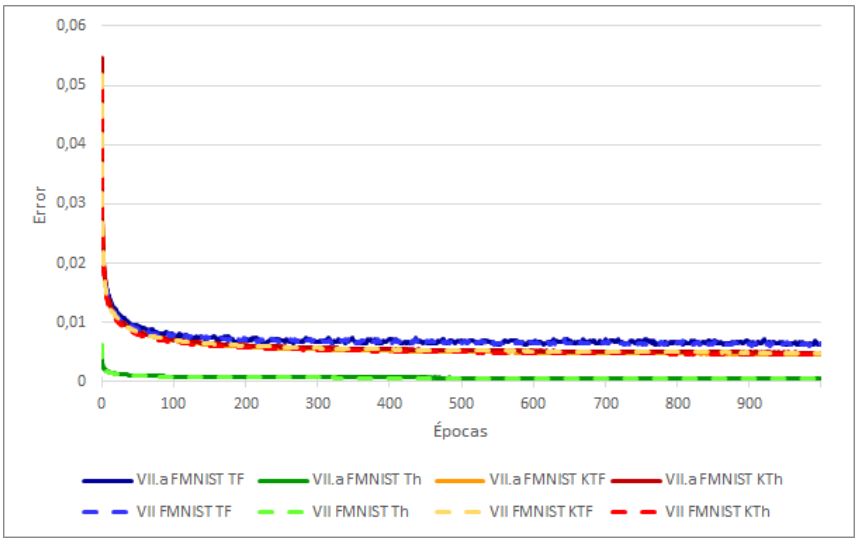



Fig. 5.30. Resultados del experimento VII.a en *Fashion-MNIST*

### 5.11.8. Experimento VII.b

| | |
|---|---|
| Tasa de aprendizaje | 0.001 |
| Tamaño del bloque (*batch*) | 500 |
| Número de épocas | 1000 |
| Inicialización de pesos | Xavier normal |
| Inicialización de *bias* | Aleatorio normal |
| Función de activación | ReLU en todas las capas excepto sigmoide en la última |
| Optimizador | Adam |

Tabla 5.17. Características principales del experimento VII.b

Al igual que se hizo en pruebas anteriores, al cambiar la inicialización de los pesos y *bias*, se ha probado a mantener para alguno de los dos la metodología previa a la modificación, en este caso la aleatoria normal, con el fin de averiguar si los resultados mejoraban o no. Se debe concluir, a la luz de las siguientes figuras, que ha ocurrido un fenómeno similar al del experimento previo, los valores obtenidos han sido muy parecidos con respecto a los de VII, y en ocasiones ligeramente mejores a ellos, aunque de forma global sean algo peores.

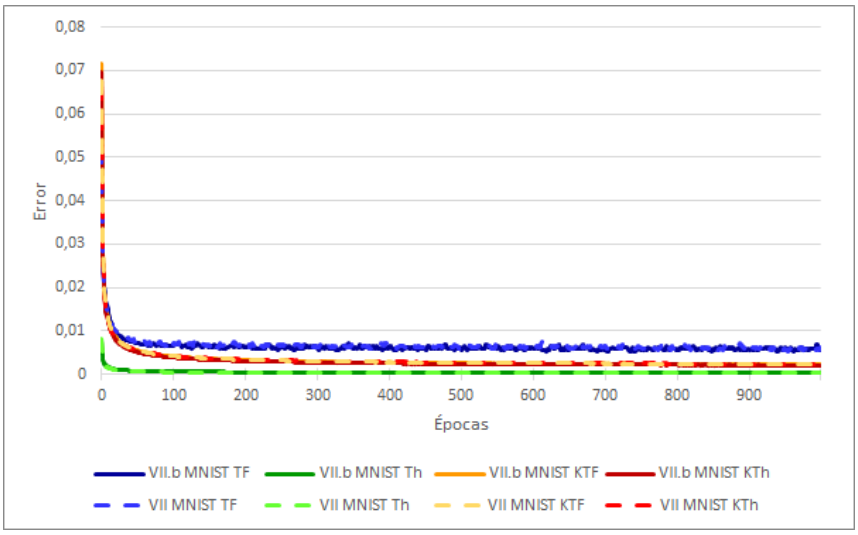



Fig. 5.31. Resultados del experimento VII.b en *MNIST*

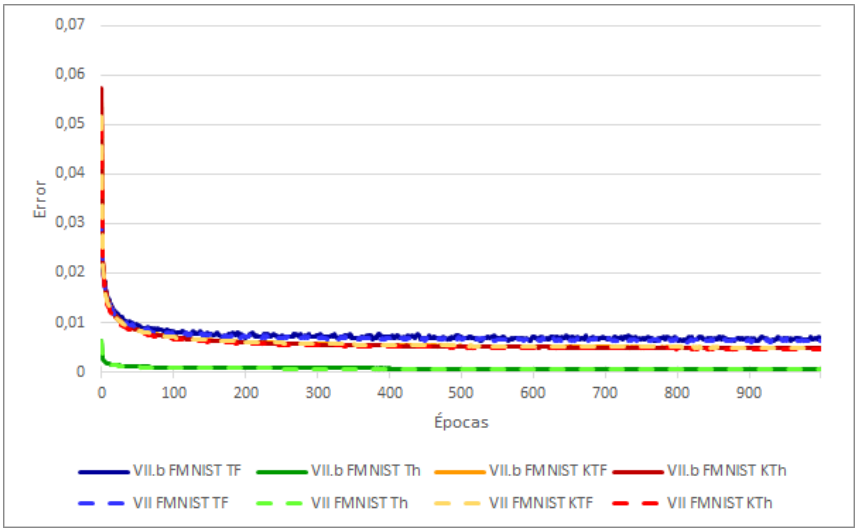



Fig. 5.32. Resultados del experimento VII.b en *Fashion-MNIST*

### 5.11.9. Experimento VII.c

| | |
|---|---|
| Tasa de aprendizaje | 0.001 |
| Tamaño del bloque (*batch*) | 500 |
| Número de épocas | 1000 |
| Inicialización de pesos | He normal |
| Inicialización de *bias* | He normal |
| Función de activación | ReLU en todas las capas excepto sigmoide en la última |
| Optimizador | Adam |

Tabla 5.18. Características principales del experimento VII.c

Además de la metodología de Xavier, para la inicialización de parámetros existe otra estrategia, similar a ella, planteada en 2015 por Kaiming He, entre otros ([89]), de la cual existen, de nuevo, dos variantes, que emplean una distribución uniforme o normal. Como de los experimentos anteriores, VII.a y IV, se concluyó que la opción uniforme ofrecía peores resultados, en este caso se ha optado directamente por la normal, con media 0 y varianza $\sqrt{\frac{2}{n_{in}}}$, donde $n_{in}$ es el número de entradas de la capa. De las siguientes figuras se puede deducir que el cambio en la inicialización ha producido un rendimiento ligeramente inferior al obtenido del experimento VII, aunque ambos son, de nuevo, muy similares.

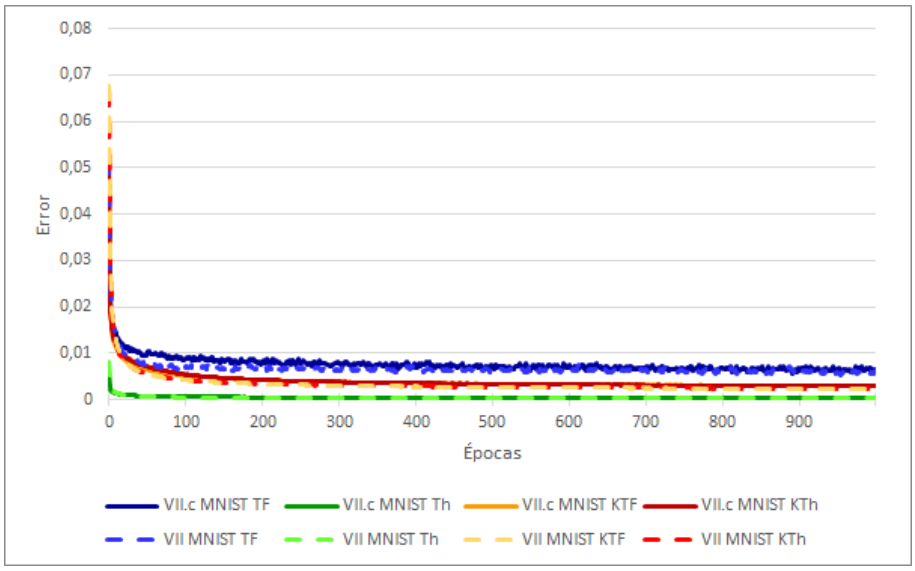



Fig. 5.33. Resultados del experimento VII.c en *MNIST*

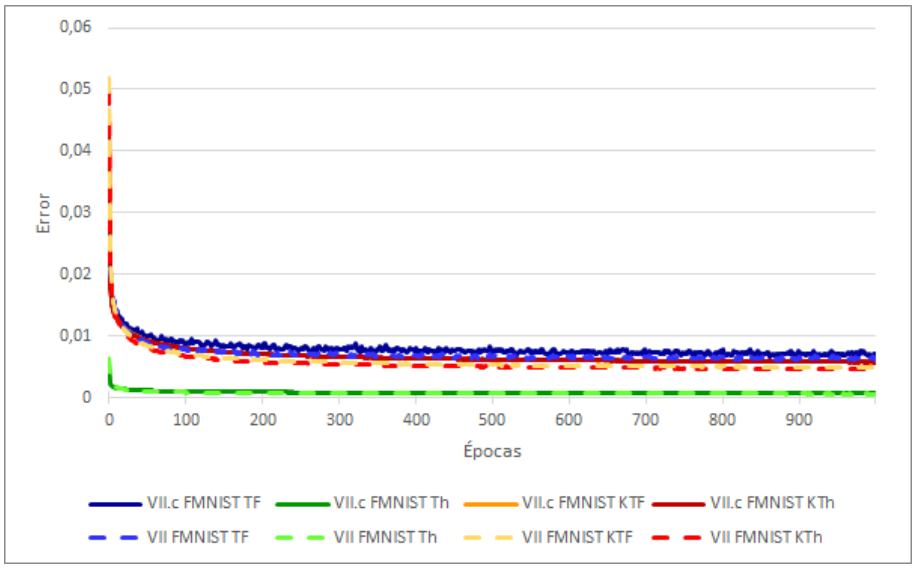



Fig. 5.34. Resultados del experimento VII.c en *Fashion-MNIST*

### 5.11.10. Experimento IX

| | |
|---|---|
| Tasa de aprendizaje | 0.01 |
| Tamaño del bloque (*batch*) | 500 |
| Número de épocas | 1000 |
| Inicialización de pesos | Xavier normal |
| Inicialización de *bias* | Xavier normal |
| Función de activación | ReLU en todas las capas excepto sigmoide en la última |
| Optimizador | SGD (*Stochastic Gradient Descent*) |

Tabla 5.19. Características principales del experimento IX

Aunque Adam sea principalmente el optimizador recomendado debido a su buen rendimiento ([90]), se han hecho pruebas con algunos más para averiguar si desempeñan un mejor papel en el caso tratado en este documento. Uno de ellos es el optimizador SGD (*Stochastic Gradient Descent*, o gradiente descendente estocástico), basado en escoger una única muestra aleatoria (o un conjunto de ellas, si se especifica el tamaño del bloque) de los datos para realizar el entrenamiento vía descenso de gradiente, por lo que el tiempo empleado para el aprendizaje se reducirá, aunque exista un mayor nivel de ruido [91]. En cuanto a la tasa de aprendizaje usada, como para Adam se seleccionó finalmente 0.001, que era la que las librerías utilizadas proponían por defecto, motivadas a su vez por las recomendaciones hechas desde el artículo en el que se presentaba dicho optimizador; para este y los subsiguientes experimentos, se emplearán los valores sugeridos por las funciones empleadas. En las siguientes figuras se puede observar que los resultados han sido claramente peores para todas las librerías, especialmente para *TensorFlow* y las dos variantes de *Keras*; en el caso de SGD, frente a Adam.

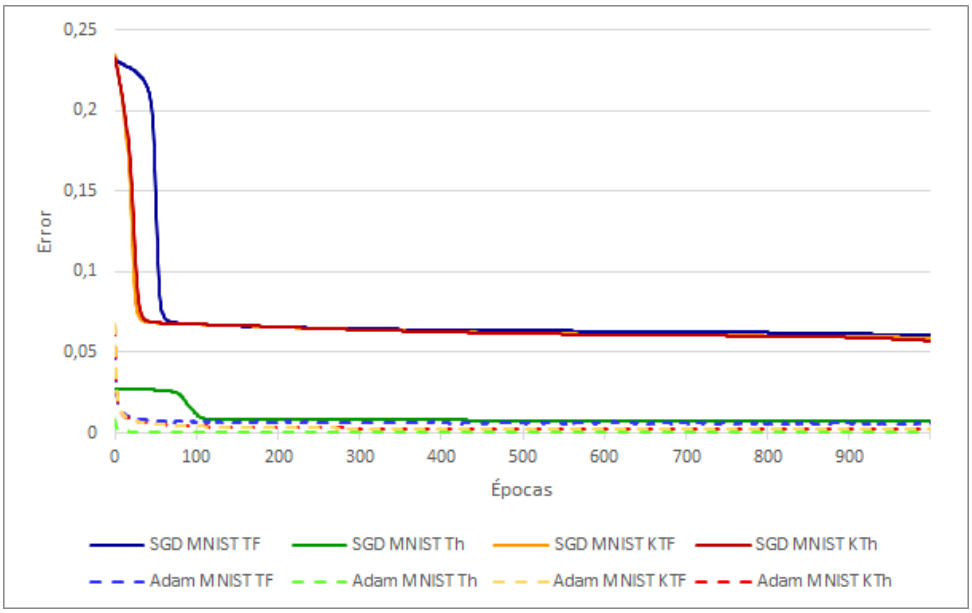



Fig. 5.35. Resultados del experimento IX en *MNIST*

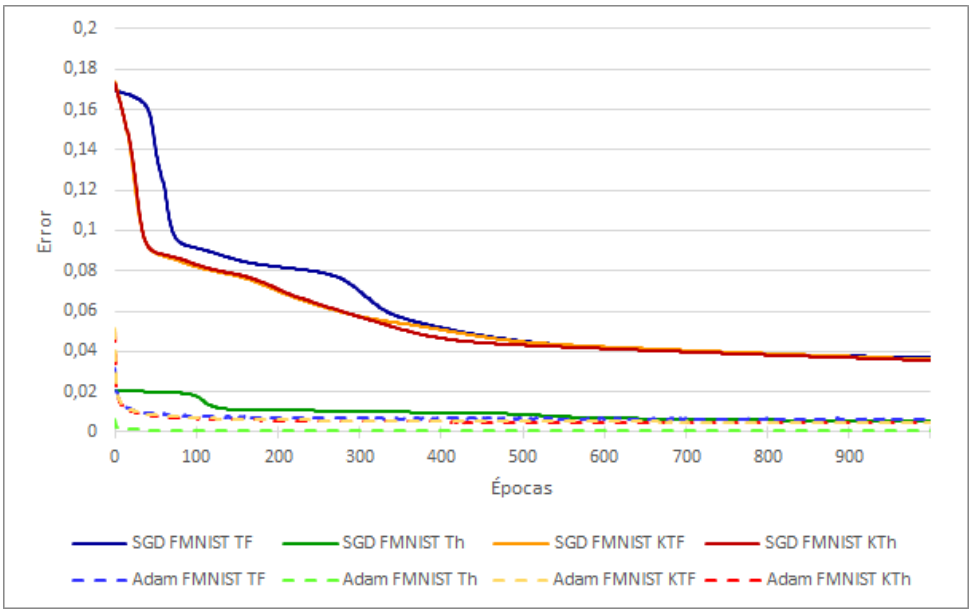



Fig. 5.36. Resultados del experimento IX en *Fashion-MNIST*

### 5.11.11. Experimento X

Además del optimizador SGD, se han realizado experimentos con el algoritmo Adagrad (*Adaptive Gradient*, gradiente adaptativo) (tabla 5.20), cuyo funcionamiento radica en almacenar la frecuencia con la que las características de los datos de entrenamiento interactúan con cada parámetro (gradiente), y adaptar a la vez la tasa de aprendizaje de cada uno de ellos, disminuyéndola más rápido para aquellos que aparecen de forma más usual, y más despacio para los menos habituales [92]. A la vista de las siguientes figu-

| | |
|---|---|
| Tasa de aprendizaje | 0.01 |
| Tamaño del bloque (*batch*) | 500 |
| Número de épocas | 1000 |
| Inicialización de pesos | Xavier normal |
| Inicialización de *bias* | Xavier normal |
| Función de activación | ReLU en todas las capas excepto sigmoide en la última |
| Optimizador | Adagrad (*Adaptive Gradient*) |

Tabla 5.20. Características principales del experimento X

ras, se puede afirmar que los resultados han sido peores para todas las librerías, de forma especialmente clara en el caso de *TensorFlow*.

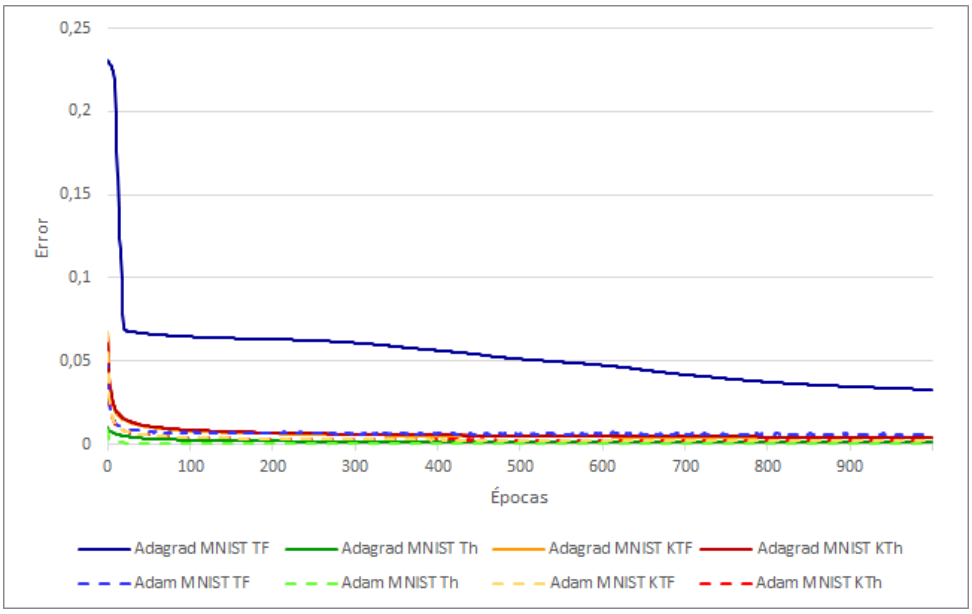



Fig. 5.37. Resultados del experimento X en *MNIST*

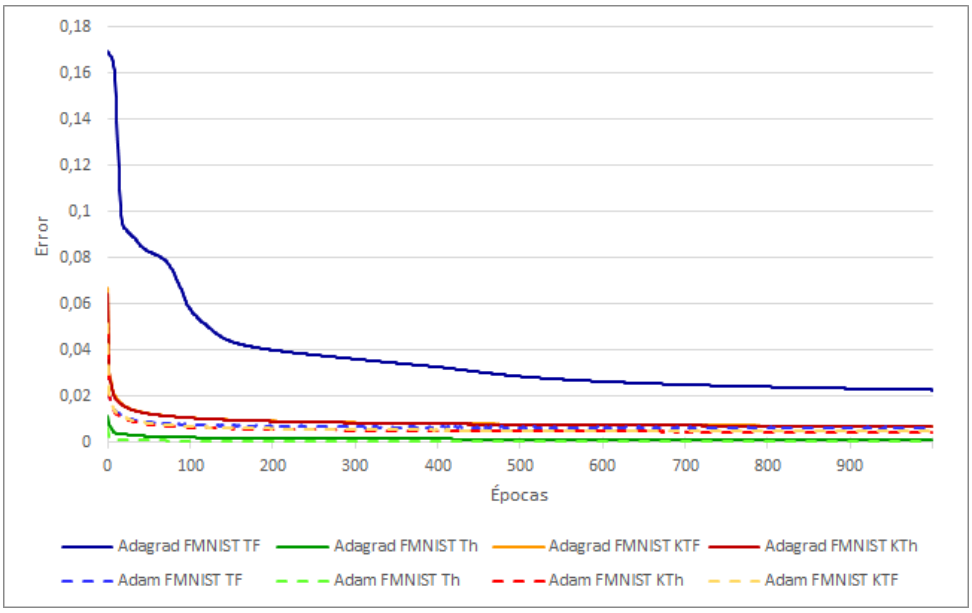



Fig. 5.38. Resultados del experimento X en *Fashion-MNIST*

### 5.11.12. Experimento XI

| | |
|---|---|
| Tasa de aprendizaje | 1.0 |
| Tamaño del bloque (*batch*) | 500 |
| Número de épocas | 1000 |
| Inicialización de pesos | Xavier normal |
| Inicialización de *bias* | Xavier normal |
| Función de activación | ReLU en todas las capas excepto sigmoide en la última |
| Optimizador | Adadelta |

Tabla 5.21. Características principales del experimento XI

El optimizador empleado en este experimento, Adadelta, surgió para intentar corregir las carencias del utilizado en el anterior, Adagrad, mediante la acumulación de las últimas frecuencias con las que los parámetros de la red han interactuado con los datos de entrenamiento (gradientes) [93]. De la misma forma que ocurría en los últimos experimentos, se da la circunstancia de que los resultados que ofrece Adam son mejores, en este caso, de manera clara para *TensorFlow* y las dos opciones de *Keras*.

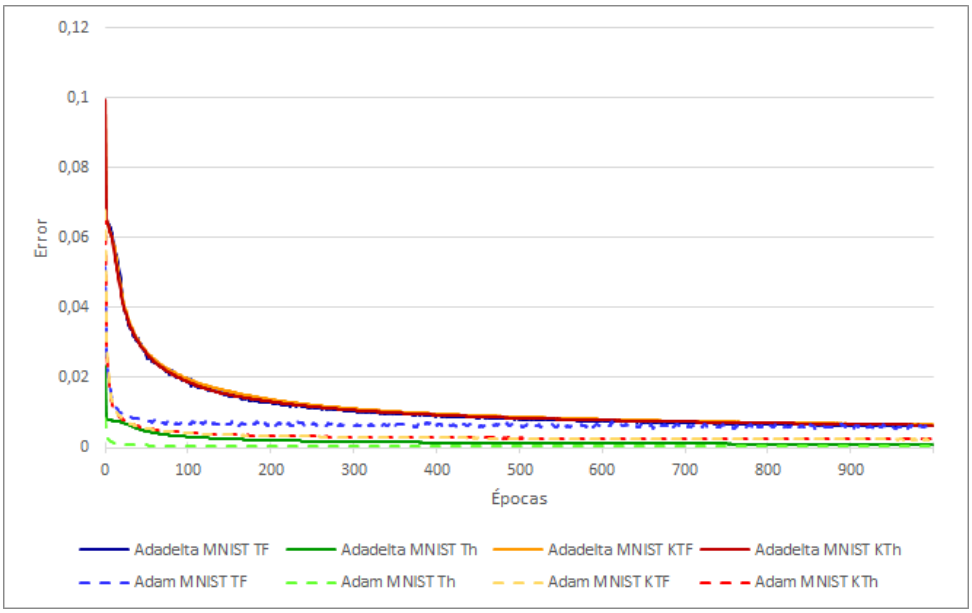



Fig. 5.39. Resultados del experimento XI en *MNIST*

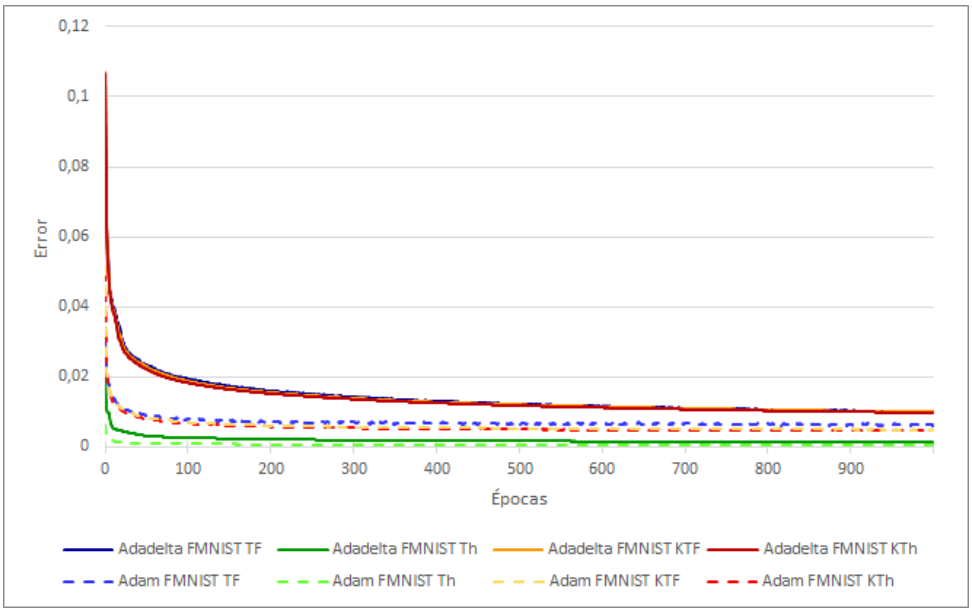



Fig. 5.40. Resultados del experimento XI en *Fashion-MNIST*

### 5.11.13. Experimento XII

En este experimento se ha utilizado el optimizador RMSprop (tabla 5.22), que también pretende corregir los posibles errores de Adagrad, y que se basa, de forma similar a Adadelta, en actualizar las tasas de aprendizaje empleando la media de los últimos gradientes al cuadrado [94]. Las gráficas siguientes continúan la tendencia de las anteriores pruebas, confirmando que con Adam se consigue un mejor rendimiento. En este caso, esa afirmación se sostiene claramente en las primeras 400 ó 500 épocas, pero luego las curvas

| | |
|---|---|
| Tasa de aprendizaje | 0.001 |
| Tamaño del bloque (*batch*) | 500 |
| Número de épocas | 1000 |
| Inicialización de pesos | Xavier normal |
| Inicialización de *bias* | Xavier normal |
| Función de activación | ReLU en todas las capas excepto sigmoide en la última |
| Optimizador | RMSProp |

Tabla 5.22. Características principales del experimento XII

de *TensorFlow* y *Keras*, principalmente, se van pareciendo más cuantas más iteraciones se produzcan.

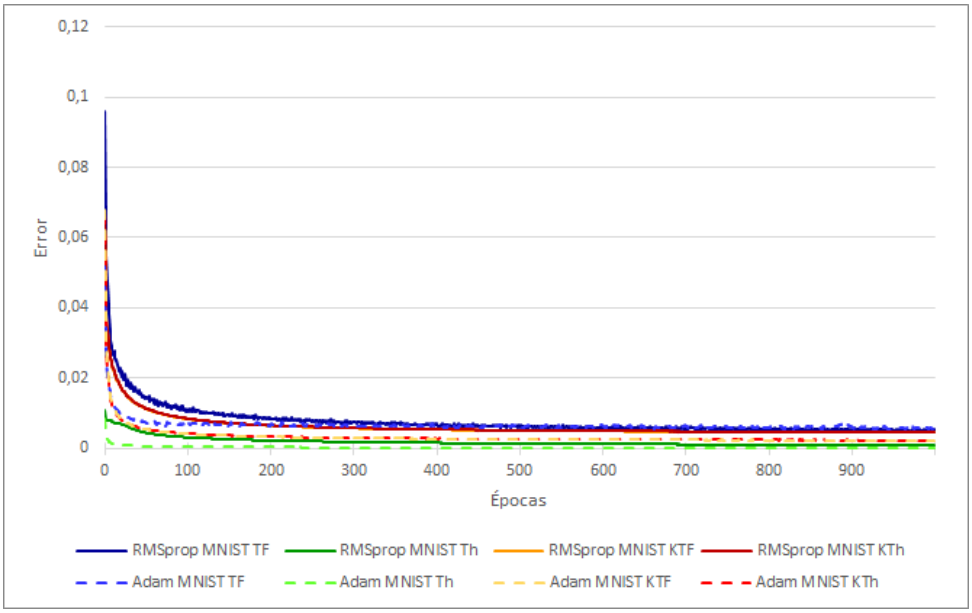



Fig. 5.41. Resultados del experimento XII en *MNIST*

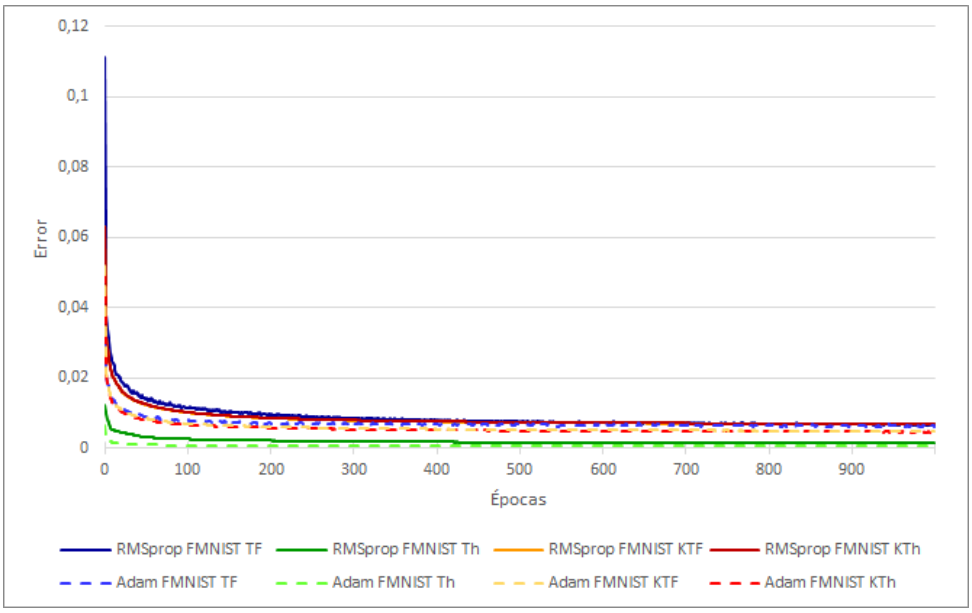



Fig. 5.42. Resultados del experimento XII en *Fashion-MNIST*

### 5.11.14. Experimento XIII

| | |
|---|---|
| Tasa de aprendizaje | 0.002 |
| Tamaño del bloque (*batch*) | 500 |
| Número de épocas | 1000 |
| Inicialización de pesos | Xavier normal |
| Inicialización de *bias* | Xavier normal |
| Función de activación | ReLU en todas las capas excepto sigmoide en la última |
| Optimizador | Adamax |

Tabla 5.23. Características principales del experimento XIII

Por último, se ha empleado un experimento con el optimizador Adamax, que fue propuesto como extensión de Adam, y que realiza la actualización mediante la norma del supremo [95] [96]. En este caso, para *TensorFlow* y *Theano* existe una implementación del algoritmo, pero debido a que no funciona correctamente y se producen errores, no se han incluido los resultados de dichas librerías. En *Keras* no se ha producido este hecho, y además los resultados han sido muy parecidos a los de Adam, llegando incluso a superarlos en algunos tramos a lo largo de las épocas.

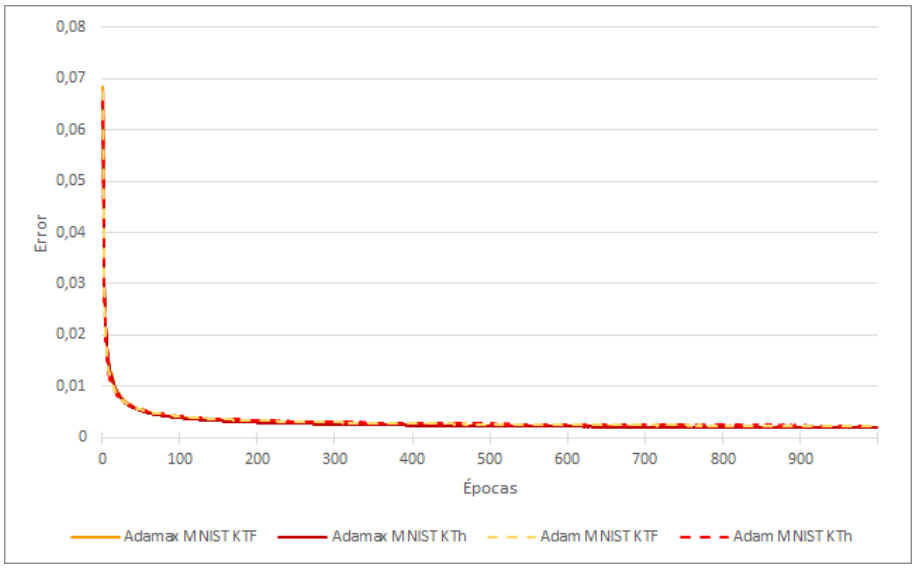



Fig. 5.43. Resultados del experimento XIII en *MNIST*

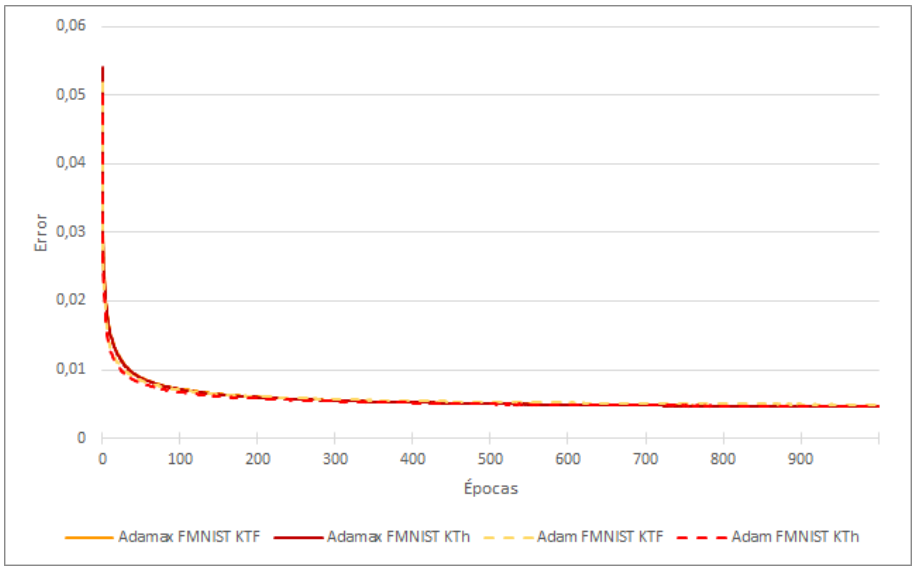



Fig. 5.44. Resultados del experimento XIII en *Fashion-MNIST*

## 5.12. Aprendizaje distribuido

A lo largo de esta sección se comentará otro de los puntos a tratar en el trabajo, la ejecución distribuida de modelos, que se realizará en *Spark* mediante una extensión de *Keras* llamada *Elephas*. Si bien la intención inicial era la de realizar esta ejecución utilizando GPU también, finalmente, por problemas con la librería utilizada, únicamente se han podido llevar a cabo con CPU, por lo que, por cuestiones de comparación, se ha efectuado además el experimento VII de esta forma.

### 5.12.1. Ejecución en CPU

En este apartado se detallarán las ejecuciones que se han llevado a cabo, y se analizarán los resultados obtenidos, así como los tiempos empleados en dichos procedimientos. Cabe destacar que los códigos utilizados se corresponden, debido a que ofrecieron el mejor rendimiento, con los del experimento VII, con la salvedad del tipo de *hardware* a emplear, que en este caso será solamente CPU, aunque su número seguirá siendo el mismo, cuatro. Es de esperar, por tanto, que al no utilizar GPU, la duración de la ejecución sea mucho mayor; mientras que los valores de error conseguidos no deberían variar, al menos no en exceso.

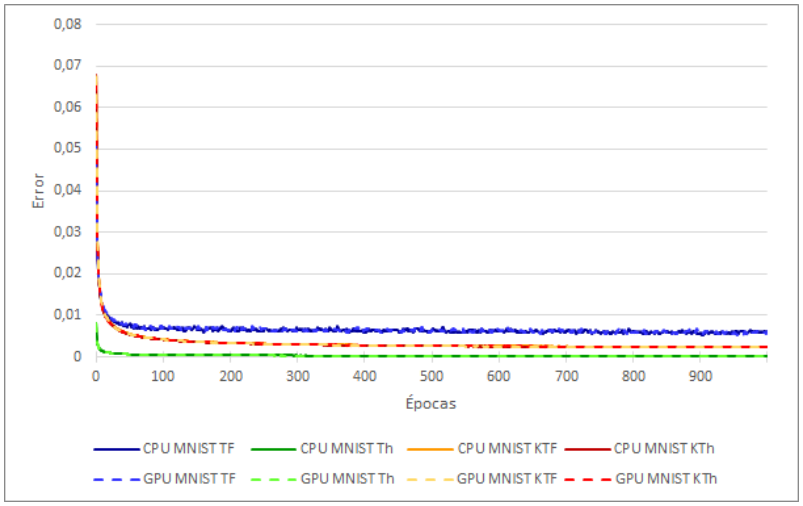



Fig. 5.45. Comparativa entre los resultados del experimento VII, utilizando CPU+GPU frente a CPU, en *MNIST*

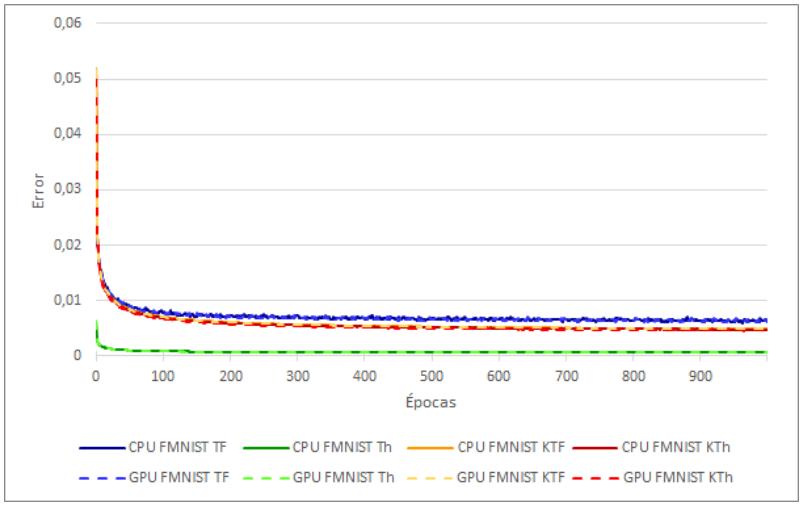



Fig. 5.46. Comparativa entre los resultados del experimento VII, utilizando CPU+GPU frente a CPU, en *Fashion-MNIST*

Observando las gráficas superiores se puede afirmar que la segunda suposición, planteada anteriormente, ha sido confirmada, ya que las curvas obtenidas mediante el uso exclusivo de CPU son prácticamente idénticas a las pertenecientes al experimento VII y, por ende, a las que empleaban además GPU.

En cuanto al tiempo, a continuación se presenta una tabla comparativa entre los dos casos, donde se puede observar cuán perjudicial ha sido la restricción en la ejecución exclusivamente a CPU, ya que la adición de la GPU supone una duración de menos del 2 % de la longitud correspondiente a su no utilización. Sobre los valores nuevos, cabe destacar que los resultados *Theano* y de *Keras* con *Theano* han resultado ser muy negativos, lo cual puede deberse a las circunstancias del experimento, como que las funciones no estén correctamente adaptadas para el uso único de CPU, las características del modelo utilizado, así como las decisiones que se han tomado para cada parámetro.

| Experimento | MNIST | | | | FMNIST | | | |
|---|---|---|---|---|---|---|---|---|
| | TF | Th | Keras TF | Keras Th | TF | Th | Keras TF | Keras Th |
| GPU+CPU | 499.59 | 538.37 | 1,118.95 | 590.60 | 502.32 | 541.11 | 1,140.26 | 590.29 |
| CPU | 57,926.20 | 352,571.79 | 68,591.34 | 267,354.78 | 54,791.47 | 337,546.89 | 65,952.47 | 268,472.80 |

Tabla 5.24. Tiempos de usuario utilizando CPU con GPU y CPU para el experimento VII, expresados en segundos.

### 5.12.2. Ejecución mediante *Elephas*

Como ya se adelantó anteriormente, para estas pruebas se utilizará únicamente el código de *Keras*, con sus dos posibles *backend*, al que se le irá modificando, para cada experimento, un conjunto de parámetros. Estos valores, que ya fueron introducidos en la explicación de las líneas que se añadían a *Keras*, son el número de núcleos (*cores*) a utilizar por máquina o ejecutor, el total de dichos núcleos que se asignarán, y la memoria máxima en cada ejecutor, que será de $4G \times n_{nucleos}$. Teniendo esto en cuenta, se plantean a continuación las siguientes pruebas:

| Experimento | Núcleos por ejecutor | Total de núcleos | Nº de ejecutores | Memoria por ejecutor |
|---|---|---|---|---|
| 4/16 | 4 | 16 | 4 | 16G |
| 4/20 | 4 | 20 | 5 | 16G |
| 5/20 | 5 | 20 | 4 | 20G |
| 4/24 | 4 | 24 | 6 | 16G |
| 6/24 | 6 | 24 | 4 | 24G |

Tabla 5.25. Experimentos realizados con *Elephas*

### Experimento 4/16

Esta prueba se ha llevado a cabo con los datos de la tabla 5.25, y del experimento VII (5.8), y se han obtenido de ella las siguientes gráficas:

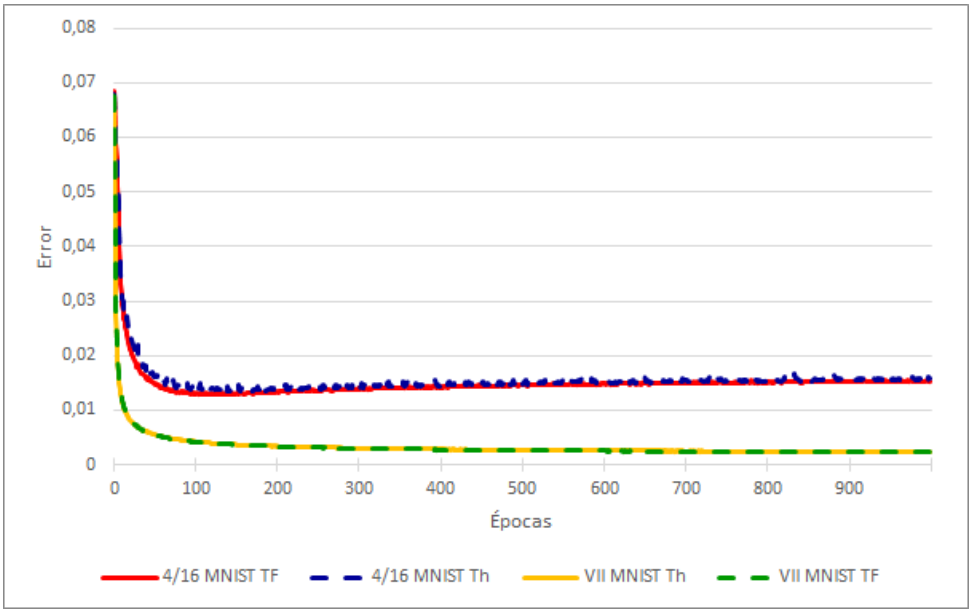



Fig. 5.47. Resultados del experimento 4/16 en *MNIST*

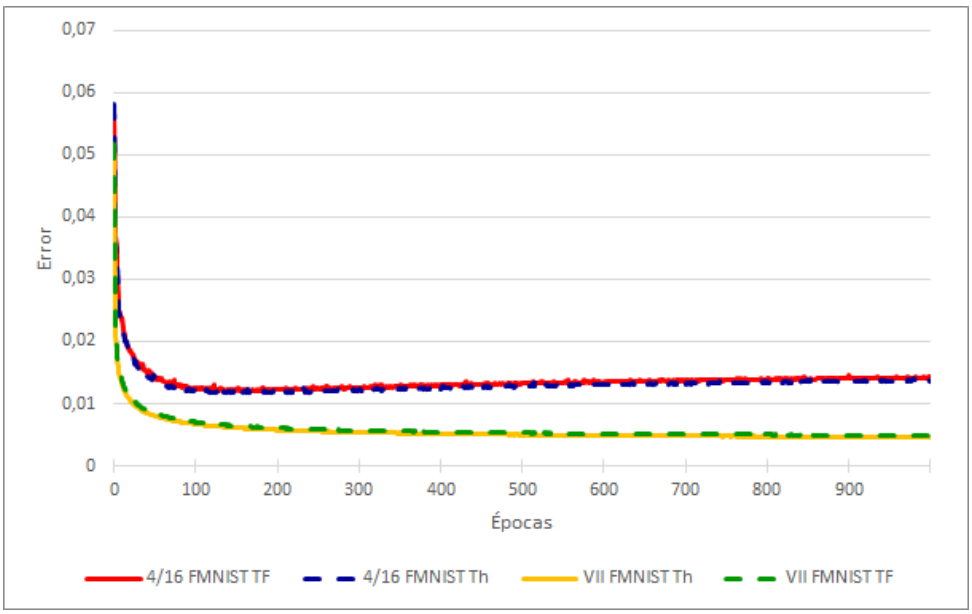



Fig. 5.48. Resultados del experimento 4/16 en *FMNIST*

Como se puede observar, los resultados han sido peores que en la ejecución lineal, y aunque no son del todo malos en comparación, el hecho de que sus valores de error aumenten con las épocas no es aceptable.

### Experimento 4/20

Este experimento fue realizado utilizando los parámetros de la tabla 5.25, y se ha comparado con los resultados del VII (5.8), obteniendo las siguientes gráficas:

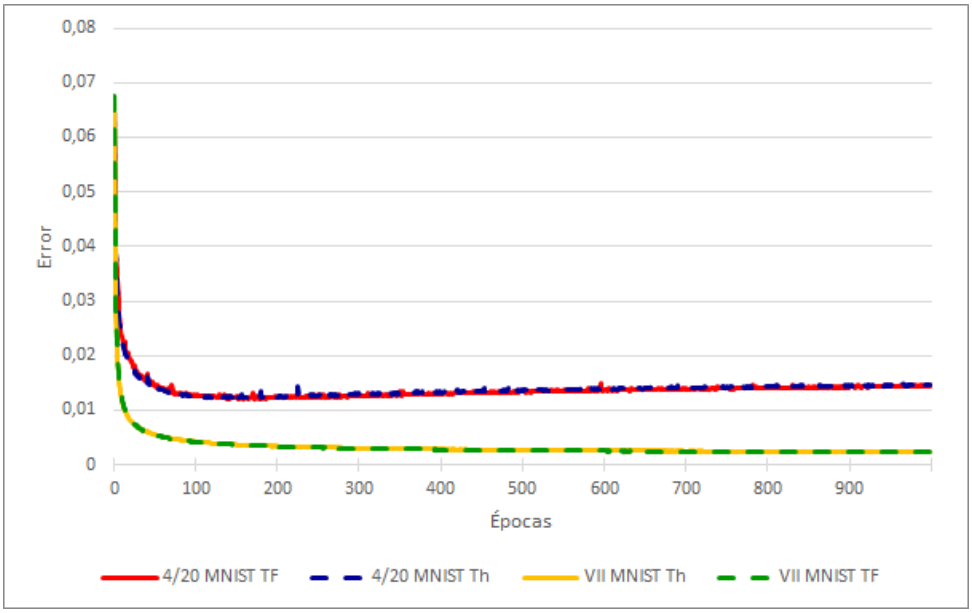



Fig. 5.49. Resultados del experimento 4/20 en *MNIST*

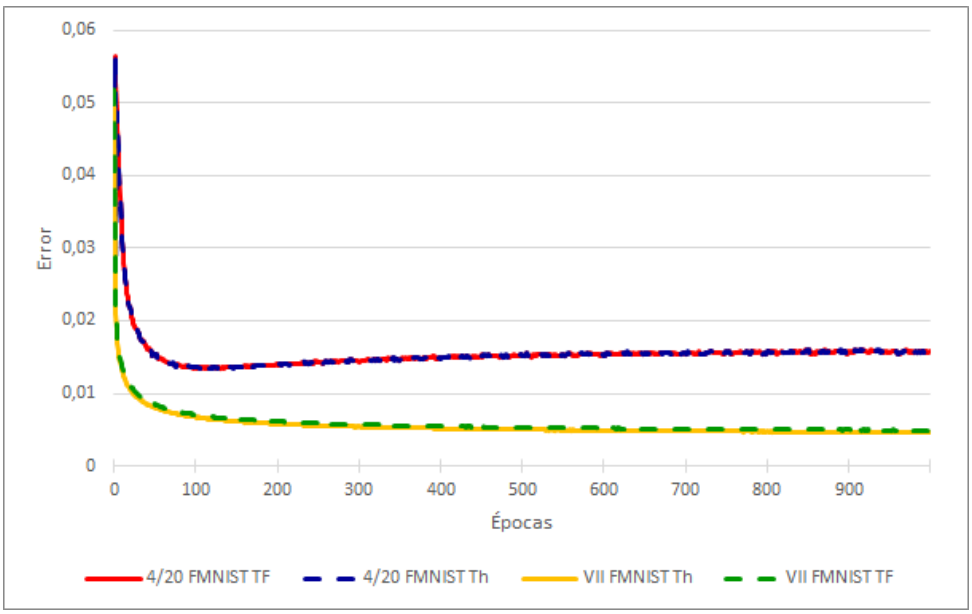



Fig. 5.50. Resultados del experimento 4/20 en *FMNIST*

Observando las figuras superiores se confirma la tendencia, planteada en el anterior experimento, de *Elephas* a empeorar el nivel de error, e incluso a ir aumentándolo, aunque los resultados finales no sean del todo malos.

**Experimento 5/20**

Para llevar a cabo esta prueba se han aplicado los valores presentados en la tabla 5.25, comparando sus resultados con los obtenidos del VII (5.8), consiguiendo las siguientes figuras:

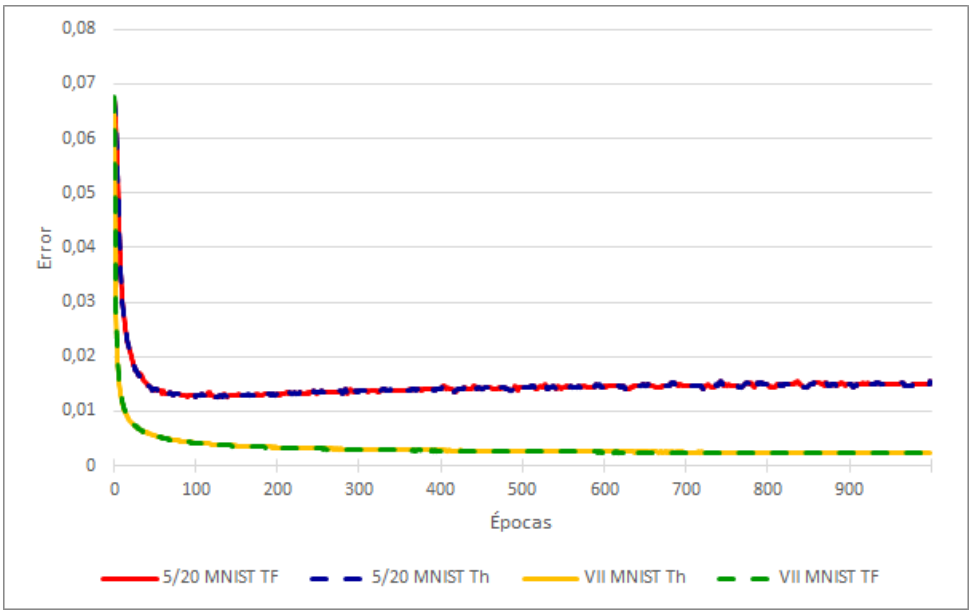



Fig. 5.51. Resultados del experimento 5/20 en *MNIST*

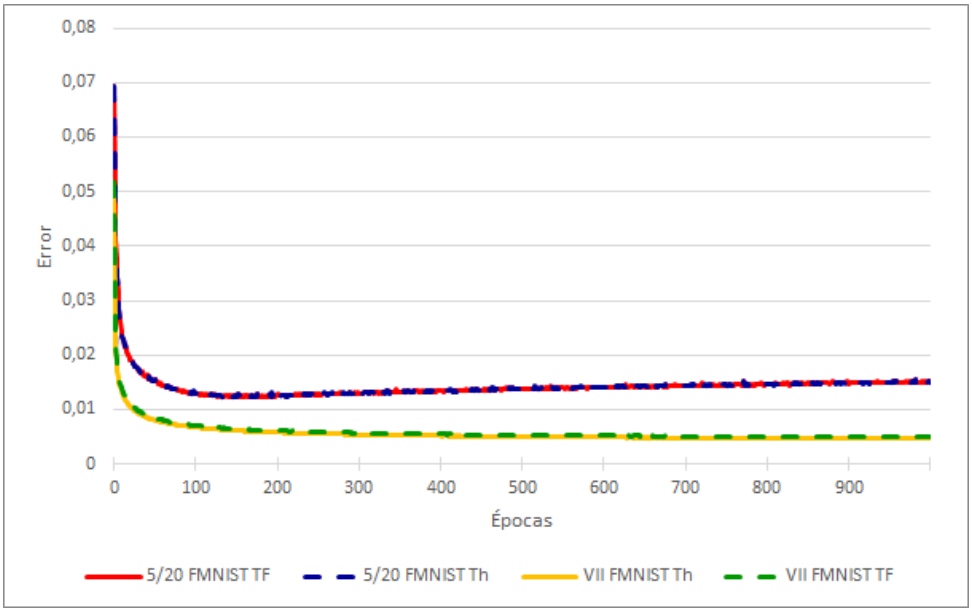



Fig. 5.52. Resultados del experimento 5/20 en *FMNIST*

De nuevo se observa en las gráficas que los valores resultantes de la adición de *Elephas*, siendo ligeramente similares a los del experimento VII, son peores en ambos casos.

**Experimento 4/24**

Este experimento se llevó a cabo utilizando los datos de la tabla 5.25, y de la comparación de los resultados obtenidos con los del VII (5.8), se consiguieron las siguientes gráficas:

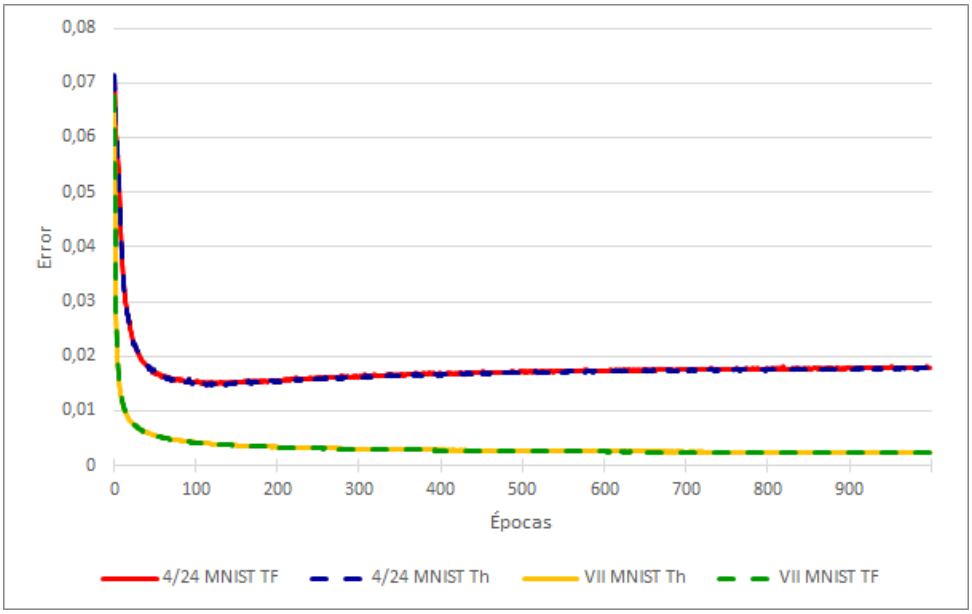



Fig. 5.53. Resultados del experimento 4/24 en *MNIST*

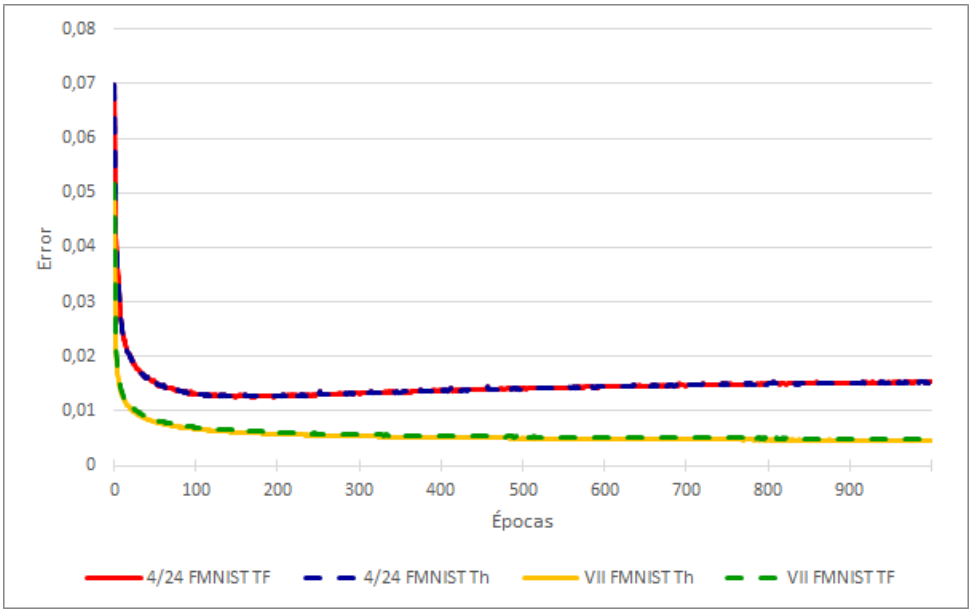



Fig. 5.54. Resultados del experimento 4/24 en *FMNIST*

En las figuras superiores se observa el mismo fenómeno que tuvo lugar en los experimentos anteriores, esto es, que añadiendo *Elephas* los resultados empeoran y que, en mayor o menor medida, el nivel de error va aumentando.

### Experimento 6/24

Esta prueba se realizó empleando los valores de la tabla 5.25, y sus resultados se han contrastado con los del VII (5.8), obteniendo las siguientes figuras:

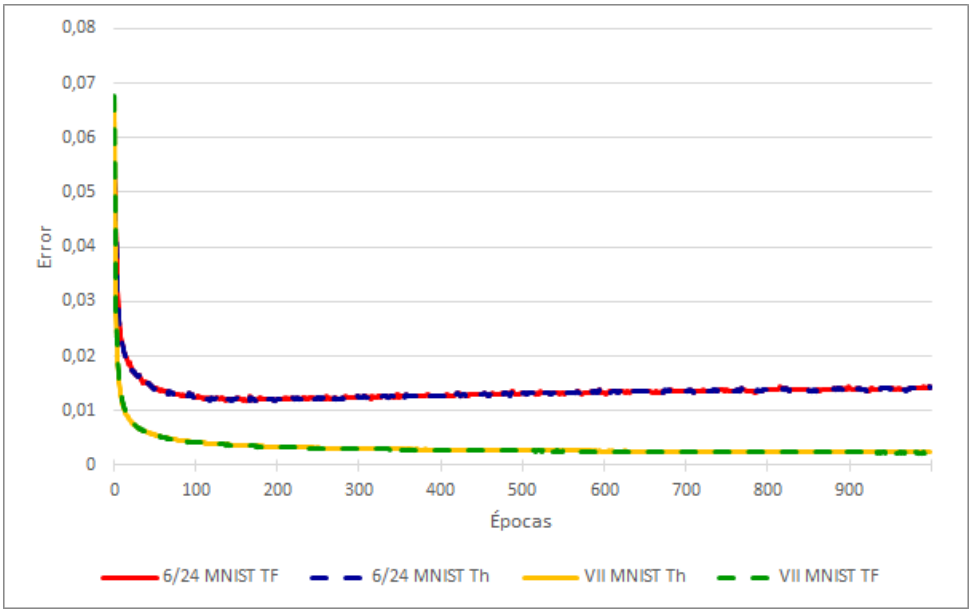



Fig. 5.55. Resultados del experimento 6/24 en *MNIST*

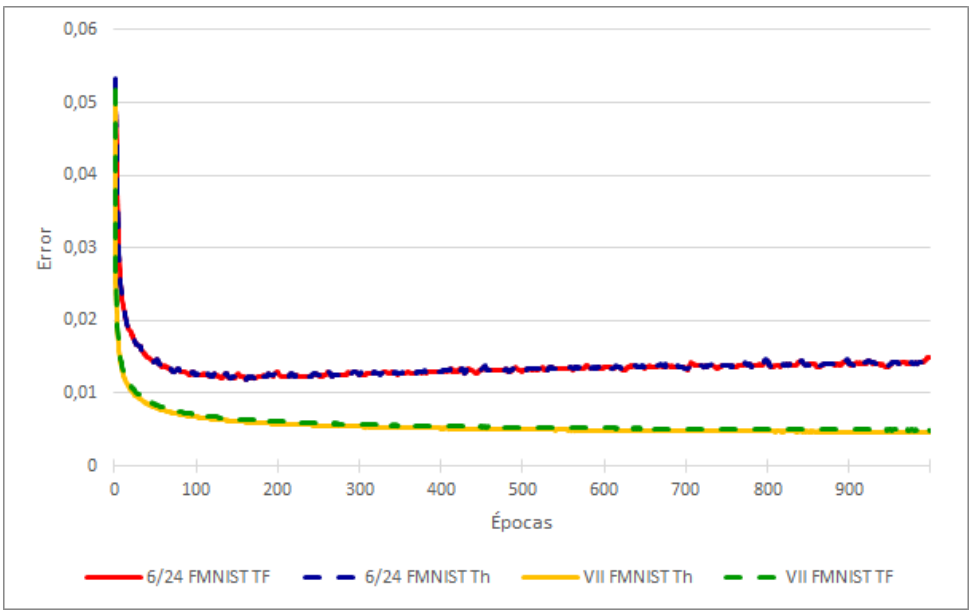



Fig. 5.56. Resultados del experimento 6/24 en *FMNIST*

Como ocurrió anteriormente, los valores de error empeoran al añadir la capa de *Elephas*, e incluso llegan a aumentar ligeramente con el transcurso de las épocas.

## 5.13. Análisis de resultados

En esta sección se realizará un análisis global de todos los experimentos que han sido realizados a lo largo del trabajo, comentando las razones que existen detrás de cada uno de ellos, así como los resultados obtenidos. En primer lugar se presenta una tabla (5.26), a la que se hará referencia a lo largo de este texto, que contiene el tiempo que cada prueba ha consumido en ejecutarse. Dichos valores han sido conseguidos mediante el comando de Linux *time*, que devuelve, entre otra información, las cifras correspondientes al tiempo real, lo que el programa ha tardado en total; del sistema, lo que las CPU han empleado en tareas relacionadas, como la asignación de memoria; y del usuario, es decir, el tiempo que los núcleos han invertido en la ejecución del proceso. Es este último dato el que se ha recopilado de cada una de las ejecuciones, que como ya se indicó anteriormente, han sido cinco por cada experimento; de las cuales se ha obtenido una media, que es la que aparece reflejada en la tabla.

| Experimento | *MNIST* | | | | *FMNIST* | | | |
|---|---|---|---|---|---|---|---|---|
| | TF | Th | *Keras* TF | *Keras* Th | TF | Th | *Keras* TF | *Keras* Th |
| I | 776.62 | 655.89 | 1889.49 | 738.22 | 776.23 | 638.79 | 1856.86 | 755.05 |
| II | 789.17 | 659.39 | 1829.65 | 726.02 | 792.08 | 648.96 | 1839.69 | 713.89 |
| III | 782.60 | 671.98 | 1808.46 | 730.50 | 777.41 | 645.76 | 1870.41 | 723.72 |
| IV | 773.39 | 658.88 | 1913.38 | 713.65 | 775.67 | 651.87 | 1963.81 | 737.19 |
| V | 483.38 | 480.29 | 1087.51 | 505.37 | 485.85 | 498.97 | 1069.71 | 533.00 |
| VI | 507.90 | 538.69 | 1097.73 | 575.12 | 499.38 | 542.71 | 1125.20 | 573.56 |
| VII | 499.59 | 538.37 | 1118.95 | 590.60 | 502.32 | 541.11 | 1140.26 | 590.29 |
| VIII | 770.91 | 554.82 | 1111.01 | 571.09 | 774.18 | 549.65 | 1082.90 | 600.26 |
| III.a | 776.13 | 648.36 | 1806.48 | 730.76 | 790.91 | 642.47 | 1907.03 | 737.11 |
| IV.a | 772.72 | 650.18 | 1785.14 | 692.08 | 775.79 | 635.93 | 1828.17 | 681.69 |
| IV.b | 727.87 | 665.72 | 1863.58 | 702.04 | 773.12 | 657.76 | 1795.06 | 696.30 |
| V.a | 308.36 | 390.85 | 538.17 | 385.22 | 294.68 | 368.86 | 547.59 | 393.74 |
| V.b | 4883.75 | 3637.92 | 12293.35 | 3812.73 | 4749.93 | 3565.53 | 12214.82 | 3886.92 |
| VI.a | 495.37 | 486.70 | 1087.60 | 515.03 | 487.58 | 496.66 | 1103.17 | 505.57 |
| VII.a | 505.26 | 541.06 | 1104.43 | 574.82 | 499.73 | 549.09 | 1095.80 | 592.88 |
| VII.b | 504.44 | 554.06 | 1113.68 | 588.63 | 490.58 | 565.07 | 1069.77 | 582.38 |
| VII.c | 505.03 | 541.51 | 1125.78 | 579.37 | 499.90 | 552.51 | 1128.62 | 593.12 |
| VIII.a | 498.95 | 558.31 | 1106.00 | 580.03 | 505.06 | 550.28 | 1121.81 | 582.30 |
| IX | 445.65 | 452.55 | 867.61 | 505.86 | 441.86 | 452.23 | 857.19 | 489.80 |
| X | 472.41 | 554.03 | 899.57 | 545.98 | 464.32 | 566.96 | 895.34 | 556.14 |
| XI | 500.66 | 552.94 | 1216.34 | 586.17 | 490.48 | 559.99 | 1273.88 | 602.19 |
| XII | 481.48 | 563.45 | 962.77 | 577.28 | 489.33 | 571.43 | 1010.57 | 557.71 |
| XIII | - | - | 995.76 | 565.96 | - | - | 992.12 | 574.72 |

Tabla 5.26. Tiempos de usuario de cada experimento, expresados en segundos

Para comenzar los experimentos, se decidió empezar con unos valores concretos para cada parámetro, que después se irían ajustando en función de los resultados obtenidos. Es el caso, por ejemplo, de la tasa de aprendizaje, cuyo primer valor escogido fue 0.1, aun suponiendo que iba a ser demasiado alto, como posteriormente se demostró gráficamente; y que acabó siendo 0.001, que de hecho es el seleccionado por defecto para el optimizador empleado. No obstante, cuando se observó que las gráficas de *TensorFlow* oscilaban demasiado, dicha cifra se redujo en un orden de magnitud, hasta 0.0001, lo cual, por otra parte, perjudicó el rendimiento del resto de librerías. Por otro lado, desde el primer experimento, el número de épocas se mantuvo constante en 1000, debido, como ya se comentó previamente, a que era lo suficientemente grande como para examinar la tendencia de los resultados, pero no demasiado, lo cual habría dilatado en exceso el tiempo de ejecución de los procesos. Además de esto, para el tamaño del bloque, o *batch*, se escogió en primer lugar 240, lo cual se encontraba dentro del rango habitual de valores, y se incrementó a 500 para intentar reducir el error, sabiendo también que el tiempo empleado en ejecutar el código disminuiría. En cuanto a la inicialización de pesos y *bias*, se comenzó por la metodología más sencilla, una distribución aleatoria uniforme, que posteriormente fue cambiada a normal, para, en último lugar, utilizar el algoritmo de Xavier; motivado siempre por la búsqueda de un menor error. Para elegir la función de activación se ha tenido una restricción, debida a la implementación del *autoencoder*, y es que esta debe retornar valores comprendidos en el rango entre 0 y 1, ya que los datos de entrada también lo estarán. Es por esto que se comenzó utilizando la función sigmoide, y posteriormente se hizo uso de ReLU (*Rectified Linear Unit*, o unidad lineal rectificada) debido a su sencillez y bajo coste computacional, que hacen que el modelo emplee menos tiempo para entrenarse, por lo que su uso suele ser muy común [97]; aunque no se utilizará en la última capa debido a su forma.

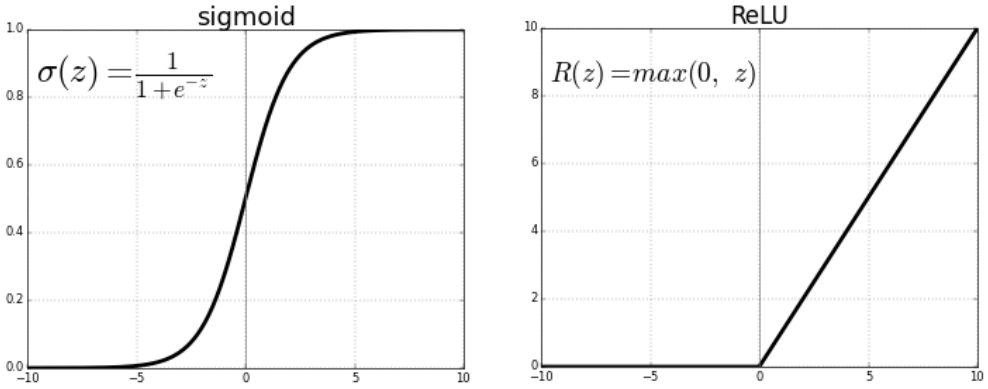



Fig. 5.57. Comparativa entre las funciones de activación sigmoide y ReLU. [98]

Para comprobar lo mencionado anteriormente, además de la función sigmoide en la última capa, se utilizó la tangente hiperbólica, cuyo límite superior también es 1, pero el inferior no es 0, sino -1. Por último, para el optimizador se ha empleado desde el principio el algoritmo Adam, ya que es uno de los más populares debido a su buen desempeño, aunque en los últimos experimentos se han realizado pruebas con otras metodologías, como SGD (*Stochastic Gradient Descent*, gradiente descendente estocástico), Adagrad, Adadelta, RMSprop o Adamax; de las cuales sólo la última ha ofrecido mejores resultados, aunque no se pudo ejecutar para todas las librerías. El funcionamiento de Adam (*Adaptive moment estimation*, o estimación adaptativa del momento), que no se aclaró anteriormente, es parecido a los demás optimizadores en cuanto a que adapta la tasa de aprendizaje de los diversos parámetros de la red en función de las estimaciones realizadas, en este caso teniendo combinando las medias ponderadas exponencialmente de los últimos gradientes, así como de sus cuadrados [96].

En cuanto a los experimentos, como ya se destacó previamente, los tres primeros consistieron en reducir la tasa de aprendizaje para conseguir que el error variase, lo que se logró en el cuarto mediante un cambio en la metodología de inicialización de pesos y *bias* por igual, conclusión que se obtuvo tras realizar varias pruebas secundarias con todas las combinaciones (véanse los experimentos IV.a y IV.b). Al margen de esto, como los resultados de *Keras* seguían sin alterarse tras el paso de las épocas, se procedió a aumentar el tamaño del bloque de entrenamiento, con lo que, al estudiar más muestras antes de actualizar los parámetros de la red, debería haber un menor ruido y, por tanto, un menor error; lo cual ocurrió únicamente en *Theano*. Paralelamente a esta quinta prueba, se realizaron otras dos, que planteaban los dos casos extremos en cuanto a la capacidad del *batch*: un valor muy pequeño, y otro muy grande. De estos experimentos se llegaron a varias resoluciones, la primera era que, como cabía esperar, un bloque de menor tamaño, al contener menos muestras y, por tanto, más ruido; funcionaría peor que uno en el que fuera mayor (sirvan como demostración las curvas de *Theano* en 5.23, 5.24, 5.25 y 5.26) en conjuntos de entrenamiento más complejos, como *Fashion-MNIST*. Además de eso, al disminuir su extensión, el tiempo de ejecución del código aumenta drásticamente, y aunque al contrario también ocurre, esa disminución no está tan acentuada, como se puede observar en la tabla de tiempos (5.26) de forma global, o a continuación en un extracto (5.27) de la misma (se recuerdan los tamaños para cada prueba: V, 500; V.a, 2000; y V.b, 32). Este cambio en la duración podría deberse a que cuanto más pequeños son, más bloques de entrenamiento se forman, con lo que se realizarán más operaciones en

conjunto, así como más accesos a los núcleos, con lo que existirá un mayor coste en cuanto a las cabeceras añadidas a la información.

| Experimento | *MNIST* | | | | *FMNIST* | | | |
|---|---|---|---|---|---|---|---|---|
| | TF | Th | *Keras* TF | *Keras* Th | TF | Th | *Keras* TF | *Keras* Th |
| V | 483.38 | 480.29 | 1087.51 | 505.37 | 485.85 | 498.97 | 1069.71 | 533.00 |
| V.a | 308.36 | 390.85 | 538.17 | 385.22 | 294.68 | 368.86 | 547.59 | 393.74 |
| V.b | 4883.75 | 3637.92 | 12293.35 | 3812.73 | 4749.93 | 3565.53 | 12214.82 | 3886.92 |

Tabla 5.27. Extracto de la tabla de tiempos de usuario para cada experimento, con todos los valores expresados en segundos.

A raíz del experimento V se llegó a otra conclusión, esta vez relacionada con el tipo de función de activación utilizado, la sigmoide, que debido a su forma y a que se emplea en todas las capas de la red, puede llegar a reducir prácticamente a cero la frecuencia con que cada neurona se activa ante ciertas características, si esta no es muy alta; y también podría hacer que el modelo no aprendiese, en el caso de que sí lo fuera [87]. Es por esto que para la siguiente prueba se planteó el uso de la función ReLU en todas las capas del *autoencoder*, exceptuando la última, con el fin de, como se indicó anteriormente, obtener valores en el rango entre 0 y 1. No obstante los resultados obtenidos (5.11 y 5.12), aunque mejoraron para *Keras*, no fueron muy buenos en *TensorFlow* y *Theano*, donde el error no varió a lo largo de las épocas. Con el objetivo de solucionar esto se planteó el experimento VII, en el que se modifica de nuevo la inicialización de pesos y *bias*, en este caso a la versión normal de la metodología de Xavier, con la que se consiguieron mejorar todos los resultados para ambos conjuntos de datos de entrenamiento. Se logró un efecto muy similar, aunque no tan bueno, en las pruebas VII.a, VII.b y VII.c, donde se utilizaron otros algoritmos, como la versión uniforme de Xavier, o la normal de He. En los cuatro casos mencionados ocurre que, para *TensorFlow*, la curva obtenida oscilaba significativamente, por lo que se planteó un último experimento, el VIII, en el que la tasa de aprendizaje se reducía en un orden de magnitud, con el fin de comprobar si dicha oscilación se mantenía, y si los demás resultados mejoraban con el cambio. A la luz de las gráficas conseguidas (véanse figuras 5.15 y 5.16), se puede afirmar que si bien el primer objetivo sí se logró, los valores hallados para el resto de librerías no mejoraron, aunque se mantuvieron muy similares. Por último, se probaron distintos optimizadores para comparar su rendimiento con el de Adam, que se había estado utilizando desde el principio, por las causas anteriormente comentadas. Si bien es cierto que ninguno de los demás algoritmos ofreció claramente unos resultados mejores, algunos, como Adamax,

se quedaron muy cerca, e incluso en algunos tramos de épocas consiguieron superarlos.

En cuanto a las ejecuciones resultantes de la adición de la capa de *Elephas*, cabe destacar que los resultados, si bien son similares a los pertenecientes al experimento VII, han empeorado ligeramente, y además en gran parte de los casos se observa claramente cómo el nivel de error va aumentando incluso. Esto puede deberse, en parte, a que al distribuir la ejecución se suele tener que sacrificar la forma en que éste se realiza, restringiendo, entre otras posibles opciones, la forma en que se manejan y tratan los datos, o el entrenamiento que se realiza sobre ellos.

Para concluir, se debe hacer una mención general a la tabla de tiempos presentada al principio del capítulo (5.26), para indicar que a nivel global, el tiempo empleado por *Theano* ha sido el más pequeño de todos, aunque elecciones en ciertos parámetros, como el aumento en el tamaño del bloque de entrenamiento, o el uso de optimizadores distintos a Adam; lo han perjudicado claramente, justo al contrario que *TensorFlow*, que sí que se ha visto beneficiado. En cuanto a *Keras*, es curioso observar cómo el tiempo correspondiente al *backend* de *Theano* ha sido de forma mayoritaria aproximadamente la mitad que el que utiliza *TensorFlow*, lo cual podría deberse al manejo de memoria, es decir, la gestión de subida y bajada de datos a la GPU. Además de esto, como cabía esperar, al emplear únicamente CPU en la ejecución, el tiempo aumenta de forma considerable, por lo que se añadió la capa de *Elephas*, mediante la cual se consiguió paliar este efecto, reduciendo la duración notablemente, como se puede deducir de la siguiente tabla:

| Experimento | *MNIST* | | *FMNIST* | |
|---|---|---|---|---|
| | *TensorFlow* | *Theano* | *TensorFlow* | *Theano* |
| 4/16 | 7,374 | 61,519 | 7,544 | 29,411 |
| 4/20 | 12,727 | 61,110 | 11,050 | 64,684 |
| 5/20 | 22,119 | 24,815 | 18,011 | 59,522 |
| 4/24 | 5,021 | 88,608 | 5,185 | 26,290 |
| 6/24 | 13,340 | 22,634 | 19,351 | 41,294 |

Tabla 5.28. Tabla de tiempos de los experimentos realizados con *Elephas*, con todos los valores expresados en segundos.

Sobre los tiempos empleados en la ejecución, cabe destacar, como sucedía al utilizar sólo CPU en el experimento VII, que *Theano* ha ofrecido un rendimiento muy inferior al de *TensorFlow*, cuyas duraciones suponen, en general, entre una tercera y una cuarta parte del total empleado por *Theano*. La razón detrás de la disparidad de resultados puede

venir dada porque *Elephas* distribuye los bloques de datos de forma desigual para todo el tiempo que dure la ejecución y no aprovecha los posibles recursos libres que puedan aparecer, por lo que puede haber ejecuciones que terminen mucho antes que otras. Esto se puede constatar en la siguiente captura de la interfaz *Apache Mesos*, en la que se ve cómo una de las tareas ya ha terminado, mientras que a las restantes aún les quedaban entorno a 50 épocas para terminar, en el momento en que fue tomada la imagen.

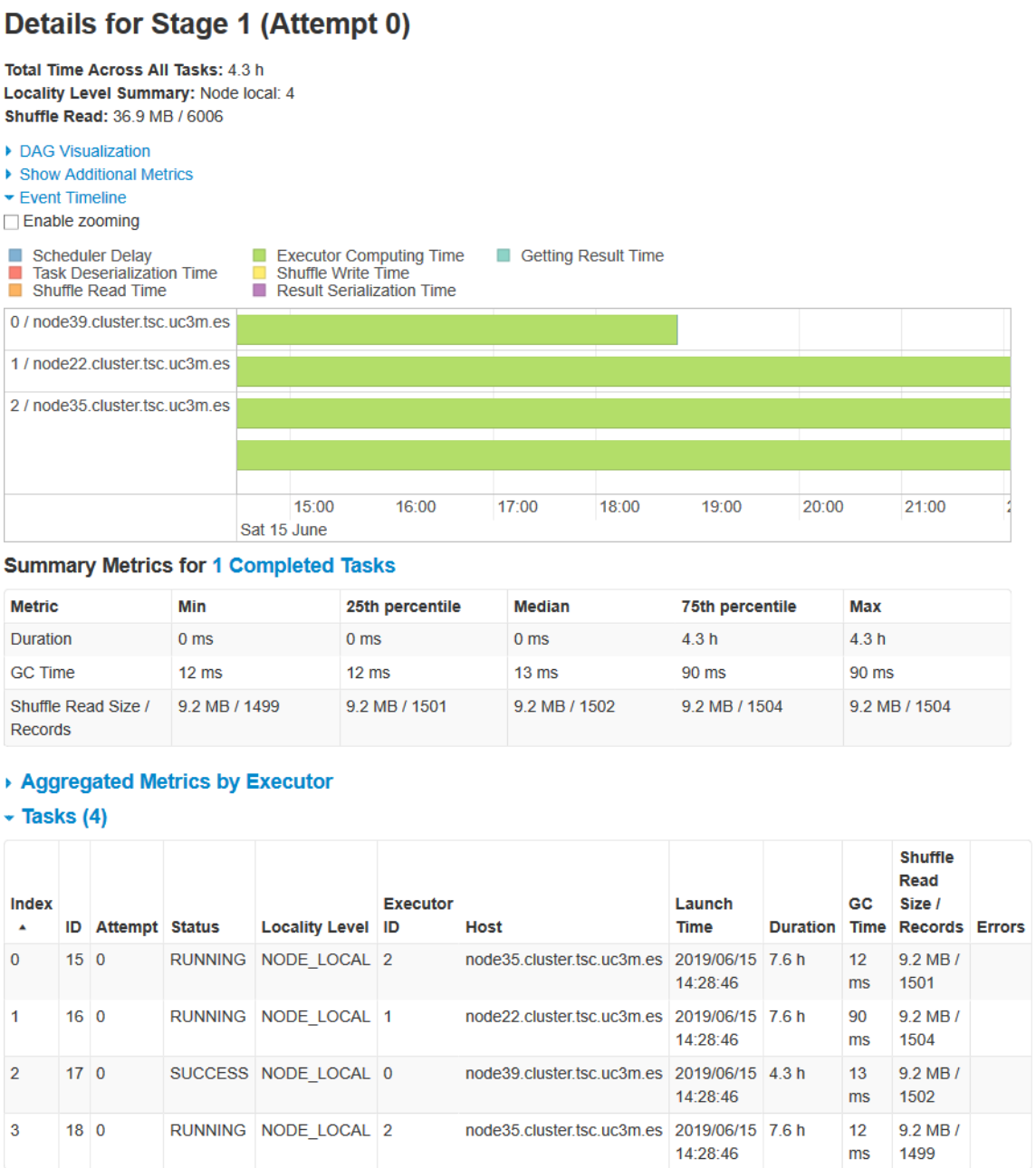



Fig. 5.58. Captura de *Apache Mesos* hacia el final de la ejecución del experimento 5/20.

Por último, se ilustra el funcionamiento de un *autoencoder* con las siguientes figuras (5.59 y 5.60), donde se observan dos reducidos grupos de imágenes, pertenecientes a los

set de datos de prueba de *MNIST* y *Fashion MNIST*, junto a sus correspondientes versiones reconstruidas mediante *TensorFlow*, *Theano*, *Keras TensorFlow* y *Keras Theano*. Cabe indicar, además, que los parámetros utilizados para esta labor han sido los correspondientes al experimento VII, el cual, como ya se ha explicado anteriormente, ha ofrecido el mejor rendimiento en cuanto a la tasa de error.

(a) Original

(b) *TensorFlow*

(c) *Theano*

(d) *Keras TensorFlow*

(e) *Keras Theano*

Fig. 5.59. Imágenes original y reconstruidas de *MNIST* en el experimento VII.

Debido a la sencillez de *MNIST*, resulta más fácil para la red aprender las características del conjunto de datos de entrenamiento, por lo que, como se advierte en la figura superior, las versiones reconstruidas no difieren prácticamente nada de las originales.

Por contra, en la siguiente imagen (5.60), correspondiente a *Fashion MNIST*, sí que se atisba una reconstrucción peor, especialmente en las prendas con multitud de detalles (por ejemplo la primera, la segunda y la novena), que dificultan su aprendizaje. Además, en este caso sí que se aprecian algunas diferencias más entre las figuras regeneradas, sin ser claramente, no obstante, ninguna de ellas mejor al resto.

(a) Original

(b) *TensorFlow*

(c) *Theano*

(d) *Keras TensorFlow*

(e) *Keras Theano*

Fig. 5.60. Imágenes original y reconstruidas de *Fashion MNIST* en el experimento VII.

CAPÍTULO 6

# CONCLUSIONES Y TRABAJO FUTURO

Para concluir el trabajo, se presenta ahora un comentario sobre las resoluciones que se desprenden de los análisis realizados en el capítulo previo. Es necesario resaltar de nuevo, como ya se ha hecho en ocasiones anteriores, que tanto lo obtenido de los experimentos, como las deducciones elaboradas a partir de ello, se deben comprender teniendo siempre en cuenta sus condiciones, como la estructura de red, los parámetros o las bases de datos que se están usando, ya que las mismas pruebas realizadas en distintas situaciones pueden llevar a deducciones diferentes.

En primer lugar, de la comparación entre los resultados obtenidos de *TensorFlow* y *Theano*, se infiere que esta última librería ha ofrecido claramente un mejor desempeño en cuanto al nivel de pérdidas logrado, no siendo así en el caso del tiempo, donde *TensorFlow*, si bien ha cosechado valores más bajos en general, se debe apuntar que esta diferencia no ha resultado ser tan significativa. Se razona, por tanto, que a pesar de ser algo más lento en algunas circunstancias, las cotas de error conseguidas por *Theano* compensan esto en gran medida.

Sobre la adición de *Keras*, a la luz de los resultados hallados, cabe destacar que sólo se ha producido una diferencia entre ambos *backend*, *TensorFlow* y *Theano*, en el caso de los tiempos de usuario, donde, como ya se indicó anteriormente, los de este último han sido, en su mayoría, aproximadamente la mitad que en la otra variante. Por tanto, al ser prácticamente iguales los valores de error, queda claro que *Theano* ha ofrecido un mejor rendimiento. No obstante, de la comparación de esta opción con las implementaciones nativas de *Theano* y *TensorFlow*, se concluye que los valores temporales han sido parecidos entre los tres casos, aunque los niveles de error de *Theano* solo siguen siendo

mejores. En cuanto a los de ambas implementaciones de *TensorFlow*, han sido en gran medida similares, por lo que a priori, para el uso de GPU no parece útil añadir una capa de programación a mayor nivel como *Keras*.

Al restringir el uso de los núcleos únicamente a CPU, se produce un fenómeno distinto al observado previamente, ya que los tiempos de *Theano*, bien de forma nativa, o bien como *backend* de *Keras*, son mucho peores que los de *TensorFlow*. Por otra parte, los valores de error para los 4 no difieren de los pertenecientes a la ejecución con GPU y CPU, por lo que en este caso sí será recomendable utilizar *TensorFlow* de forma nativa, y no una capa como *Keras*, cuyo uso no reportará beneficios significativos.

En el caso de *Elephas*, sí que es necesario argumentar a favor de *Keras* el hecho de que admita extensiones como la mencionada, a través de las cuales se puede realizar un entrenamiento distribuido mediante *Spark*. Esta mejora influye, como era de esperar, de forma muy positiva en el tiempo de ejecución, aunque se ha observado que los resultados son ligeramente peores, e incluso que el error aumenta a través de las épocas, por lo que para decidir su uso se deberá realizar una valoración de qué ámbito es el más relevante, y en cuál se puede permitir una penalización. Ha de tenerse en consideración igualmente, que mediante esta paralelización se puede manejar una gran cantidad de datos, lo cual, dado que los conjuntos utilizados no poseían tal tamaño, no ha podido ser evaluado en este documento.

Para concluir, y recalcando de nuevo que las conclusiones y los análisis aquí planteados se deben circunscribir únicamente a los parámetros, modelos y valores seleccionados; se debe destacar que *Theano* ofrece el mejor rendimiento en cuanto a niveles de error se refiere. En el caso de no utilizar GPU en la ejecución, se deberá valorar si resulta compensatorio el uso de extensiones que permitan la paralelización del problema en cuestión, reduciendo así enormemente el tiempo empleado, aunque esto conlleve una penalización en las pérdidas ocasionadas.

Como trabajo futuro quedaría estudiar la aplicación de *Elephas* a ejecuciones mediante GPU, si los problemas existentes que impideron hacerlo en el presente trabajo se llegaran a enmendar, así como la realización de un análisis más amplio, que cubriese otras topologías de red, distintos parámetros, como optimizadores, metodologías de inicialización o funciones de activación; nuevas bases de datos, e incluso aplicaciones más concretas de los *autoencoders*, como los *variational autoencoders* o los *denoising autoencoders*.

# Anexos

ANEXO A

# MARCO REGULADOR

El auge del *Big Data* y de la ciencia de datos en los últimos años ha propiciado un aumento en la concienciación de la sociedad sobre la protección de su información que, entre otros factores, ha requerido de una regulación al respecto.

En cuanto a Europa, el Reglamento General de Protección de Datos [99] fue aprobado en abril de 2016, aunque se aplicó a partir de mayo de 2018 para dar tiempo a empresas y otros organismos a adaptarse a la nueva norma. Este código pretende dotar a los ciudadanos europeos de un mayor control sobre sus propios datos, ofreciéndoles información transparente sobre el uso que se hará de ellos, informándoles diligentemente en cuanto se sepa que estos han sido pirateados, y concediéndoles el derecho de supresión o *al olvido*. Por otro lado, en lo que a las empresas se refiere, se propone una armonización en las normas a cumplir en toda la Unión Europea, independientemente de si la sede de la compañía se encuentra dentro o no; la creación de un Delegado de Protección de Datos en aquellas entidades que procesen una gran cantidad de datos y nuevos métodos de refuerzo de la privacidad como el cifrado.

En el caso de España, en diciembre de 2018 se derogó la Ley Orgánica 15/1999 de Protección de Datos de Carácter Personal [100], para sustituirla por la Ley Orgánica 3/2018 de Protección de Datos Personales y garantía de los derechos digitales [101], que sirve a su vez como adaptación del previamente mencionado Reglamento General de Protección de Datos de la Unión Europea. Esta ley consta de los siguientes títulos:

- Título I. Disposiciones generales: se concreta el objetivo de la ley, es decir, adaptar el reglamento europeo anteriormente citado al caso español, así como su aplicación

en relación a otros artículos ya existentes.

- Título II. Principios de protección de datos: establece las características que deben existir en la seguridad de la información: exactitud, actualidad, confidencialidad, consentimiento...

- Título III. Derechos de las personas: describe los privilegios que tendrán los ciudadanos en cuanto a transparencia, acceso, rectificación o supresión de los datos.

- Título IV. Disposiciones aplicables a tratamientos concretos: comprende casos específicos, como la información financiera, de contacto, de videovigilancia o de exclusión publicitaria.

- Título V. Responsable y encargado del tratamiento: dispone los deberes que tiene aquel individuo a cargo del uso de la información, al tiempo que introduce la figura del Delegado de Protección de Datos (DPD), que deberá ser asignado en una serie de entidades, como universidades, centros sanitarios, distribuidores de energía, operadores...

- Título VI. Transferencias internacionales de datos: regula los casos en que se deba producir la entrega de información a otros países.

- Título VII. Autoridades de protección de datos: establece las bases de la Agencia Española de Protección de Datos, autoridad que vela por el cumplimiento de esta ley, así como de otros organismos autonómicos dedicados al mismo fin.

- Título VIII. Procedimientos en caso de posible vulneración de la normativa de protección de datos: precisa cómo deberán ser los trámites en el evento de que se reclamen infracciones de esta norma.

- Título IX. Régimen sancionador: define quiénes están sujetos a ser castigados en el caso de que exista un quebrantamiento de la ley, y clasifica los delitos según su gravedad.

- Título X. Garantía de los derechos digitales: reconoce un conjunto de privilegios, tales como la neutralidad de la red, el acceso universal a Internet, la seguridad digital, la protección de los menores y el derecho al olvido y a la intimidad ante dispositivos de grabación y geolocalización en el ámbito laboral; entre otros muchos.

ANEXO B

# ENTORNO SOCIOECONÓMICO

En este capítulo se detallará el ámbito que rodea la tecnología planteada en este trabajo, ya sean las redes profundas en general, o los *autoencoders* en particular; en sus vertientes social y económica. Además de esto, se indicará la planificación que se ha seguido para la realización del proyecto, así como el presupuesto que se desprende del mismo.

## B.1. Impacto socioeconómico

La revolución tecnológica presente en la sociedad tiene como uno de sus puntales más relevantes la inteligencia artificial, asociada a otros avances como el *Big Data*, las redes neuronales o el 5G. Estas técnicas poseen tal trascendencia que son capaces de modificar la sociedad, en ámbitos tan diversos como el transporte, la salud, las comunicaciones o las finanzas; aunque como herramientas, la finalidad de su uso es la que definirá el tipo de impacto que produzcan.

Por un lado, una de las aplicaciones más punteras en el uso de las redes profundas es el gran sistema de vigilancia de China, basado en el reconocimiento facial de imágenes procedentes de un número estimado de 200 millones de cámaras de seguridad, aunque se cree que en 2020 ya serán casi 300 millones [102]. Los contratos del gobierno chino, en consecuencia, están estimulando la investigación y el desarrollo de estas tecnologías, como es el caso de la policía, que tiene previsto gastar 30 billones de dólares en los siguientes años; lo que está haciendo que las compañías de este país se pongan a la cabeza

del liderazgo científico mundial. Un ejemplo de esto es que 18 países, entre los que se encuentran Ecuador, Emiratos Árabes Unidos, Pakistán, Kenia e incluso Alemania; emplean sistemas inteligentes de monitorización de origen chino [103]. Si bien es cierto que estos métodos de control han ayudado a capturar criminales y a reducir el número de delitos, cabe reflexionar sobre la enorme pérdida de privacidad que suponen, además de los posibles usos secundarios, e incluso ilegítimos, como la represión política, que los responsables de estas tecnologías pueden hacer de ellos.


Fig. B.1. Demostración de uso del software de vigilancia Skynet de la empresa china Megvii [102]

Por otro lado, al margen de los sectores de aplicación que se han mencionado previamente, y sobre los que se profundizó en capítulos anteriores, existen ámbitos, a priori menos evidentes, en los que las redes neuronales pueden ser de utilidad, como el cultural o artístico, donde han surgido proyectos entre los que se encuentra *Image Style Transfer Using Convolutional Neural Networks* [104] que, empleando *autoencoders*, entre otros tipos de redes profundas, consigue trasladar el estilo pictórico de una obra al contenido de una imagen (figura B.2).

Fig. B.2. Resultados de combinar una fotografía con varias obras de arte conocidas [104].

Google, por su parte, tiene una sección, *Arts & Culture*, que comprende proyectos como *DeepDream*, un algoritmo que genera imágenes sobreprocesando características erróneamente detectadas, dotándolas de una apariencia alucinógena [105]; *Draw to Art*, que toma un boceto realizado sobre una pantalla, y muestra obras relacionadas con él [106]; *X Degrees of Separation*, en el que se eligen dos creaciones (cuadros, construcciones, objetos...), y se muestran otros tantos que están relacionados entre sí para conectar las dos selecciones [107]; o *Life Tags*, que organiza en etiquetas las más de 4 millones de fotografías de dicha revista utilizando el aprendizaje profundo [108].


Fig. B.3. Ejemplo de uso del algoritmo *X Degrees of Separation* [107].

Fig. B.4. Ejemplo de aplicación de *DeepDream* a una imagen [109].

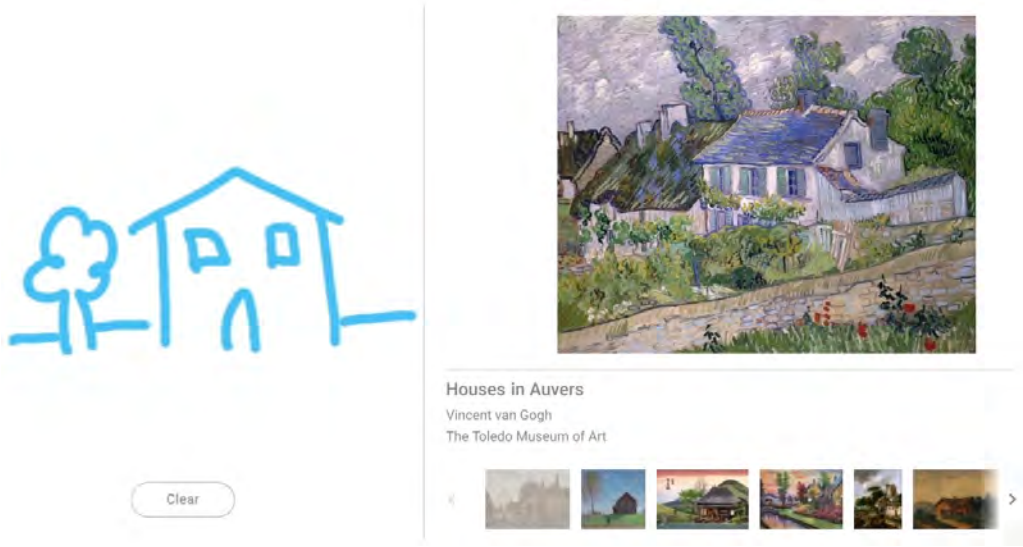



Fig. B.5. Demostración del funcionamiento de *Draw to Art* [106].

Estas iniciativas de Google, basadas en el aprendizaje profundo, no sólo suponen un gran avance en el mundo de la cultura y su preservación, sino que también inciden en ámbitos como la educación y ayudan a acercar el arte a la sociedad.

## B.2. Planificación y presupuesto del proyecto

En esta sección se presentará la planificación detallada que se ha seguido para la elaboración del proyecto, además del presupuesto asociado al mismo.

### B.2.1. Planificación del proyecto

La realización del presente trabajo se ha organizado en diferentes tareas, a desarrollar a lo largo de las semanas según la tabla B.1. Cabe destacar que existen algunas que, al poder llevarse a cabo en paralelo, se superponen, como se observará posteriormente en el diagrama de Gantt (B.6).

| **Fase 1: Estudio sobre la tecnología a utilizar** | |
|---|---|
| 1.1 Redes neuronales y aprendizaje profundo | 3 semanas |
| 1.2 *Autoencoders* | 1 semana |
| 1.3 Herramientas de implementación | 3 semanas |
| Duración de la fase 1 | 7 semanas |
| **Fase 2: Estudio previo sobre el proyecto** | |
| 2.1 Motivación y objetivo del trabajo | 1 semana |
| 2.2 Estado del arte | 2 semanas |
| 2.3 Análisis del problema | 1 semana |
| 2.4 Posibles implementaciones de la solución | 1 semana |
| Duración de la fase 2 | 5 semanas |
| **Fase 3: Diseño de la solución** | |
| 3.1 Implementación en *TensorFlow* | 1 semana |
| 3.2 Implementación en *Theano* | 1 semana |
| 3.3 Implementación en *Keras* | 1 semana |
| 3.4 Implementación en *Elephas*/*Spark* | 2 semanas |
| Duración de la fase 3 | 5 semanas |
| **Fase 4: Realización de los experimentos** | |
| 4.1 Experimentos principales | 3 semanas |
| 4.2 Otros experimentos | 2 semanas |
| 4.3 Análisis de resultados | 2 semanas |
| Duración de la fase 4 | 7 semanas |
| **Fase 5: Elaboración de la memoria** | |
| 5.1 Redacción de la memoria | 10 semanas |
| Duración de la fase 5 | 10 semanas |
| **Duración total del proyecto** | |
| | 34 semanas |

Tabla B.1. Planificación de las partes del proyecto y su duración.

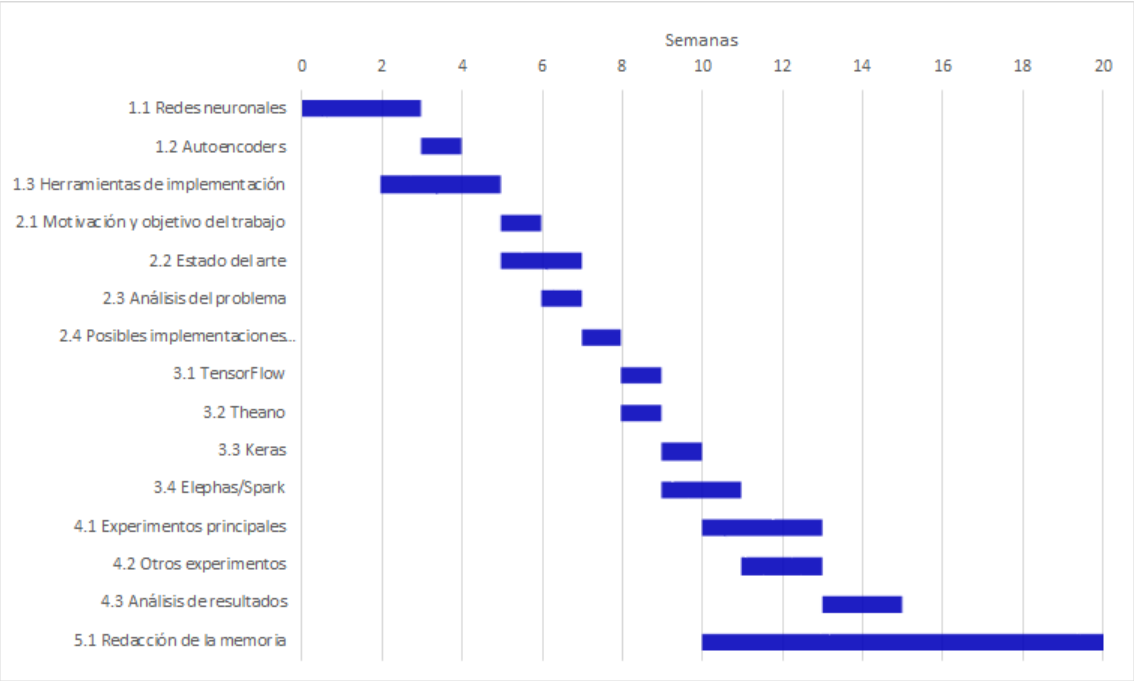



Fig. B.6. Diagrama de Gantt de la planificación.

En el diagrama de Gantt superior (B.6) se observa la forma en que las diferentes tareas han sido ejecutadas, lo cual es relevante ya que al efectuarse en paralelo se ha conseguido reducir la duración total del proyecto de 34 semanas a 20.

### B.2.2. Presupuesto del proyecto

A continuación se detallarán los conceptos en los que se desglosa el presupuesto del proyecto:

- Trabajo del alumno: considerando que el estudiante está empleado como becario, pues no ha obtenido aún su título de grado, y estimando que ha dedicado alrededor de unas 40 horas a la semana a lo largo de 4 meses, con lo que se obtiene un total aproximado de 850 horas, a la que corresponderá una asignación de 6 euros por hora.

- Trabajo del tutor: el profesor, como doctor en Ingeniería de Telecomunicaciones, se calcula que destina unas 150 horas en total a tareas de tutoría, dotadas con 24 euros por hora.

- Coste del entorno de simulación: en este apartado se encuentran los gastos relacionados con las licencias del *software* utilizado, así como los relativos al manteni-

miento y funcionamiento de los sistemas de simulación.

| **Mano de obra** | |
|---|---|
| Trabajo del alumno | 5,100 euros |
| Trabajo del tutor | 3,600 euros |
| Coste de la mano de obra | 8,700 euros |
| **Entorno de simulación** | |
| Ordenador | 500 euros |
| Uso de las granjas de simulación | 3,000 euros |
| Licencia de uso de *TensorFlow* | 0 euros |
| Licencia de uso de *Theano* | 0 euros |
| Licencia de uso de *Keras* | 0 euros |
| Licencia de uso de *Spark* | 0 euros |
| Licencia de uso de Python | 0 euros |
| Licencia de uso de Microsoft Office | 99 euros |
| Licencia de uso de *TeXworks* | 0 euros |
| Coste del entorno de simulación | 3,599 euros |
| **Coste total del proyecto** | |
| | 12,299 euros |

Tabla B.2. Presupuesto desglosado del proyecto.

Sobre el uso de las granjas de simulación, se ha realizado una estimación del coste aproximado que supondría la contratación en *Amazon Web Services* de un sistema similar, el EC2 P3 [110], la cual ascendería a más del doble (6,400 euros) del tasado para las dos granjas que finalmente se utilizaron.

ANEXO C

# RESUMEN DE CONTENIDOS EN INGLÉS

Debido al interés suscitado a nivel internacional por campos como el aprendizaje automático, la inteligencia artificial y las redes neuronales, este capítulo del anexo contiene un resumen en inglés de los contenidos presentes en el trabajo, con el fin de hacer más sencillo su acceso a estudiantes e investigadores de otros países. Comprende, por tanto, las principales secciones que componen el proyecto, como el estado del arte, los experimentos realizados y el análisis de los resultados obtenidos, así como una conclusión a modo de cierre.

El resumen de los contenidos se presenta a partir de la siguiente página.

# PERFORMANCE ANALYSIS OF AUTOENCODERS IN BIG DATA ENVIRONMENTS

## C.1. Introduction

The technological revolution in which the society has been immersed in recent years has resulted in the rise of new advances, such as data science or machine learning, that existed before, but whose potential was yet to be discovered. Those fields were relaunched as soon as the available technology was able to fulfill their needs in terms of computational capacity and speed.

The interest in these specialities was increased also with the emergence of the latest techniques and concepts, as for example Big Data and the fifth generation of cellular networks (5G), that see their own application areas strongly influenced with machine learning and neural networks, in a society in which every single device will be connected to each other, transmitting a great amount of information, and shaping an interconnected world.

However, along with the significance of these lines come numerous tools to work with and implement them in a whole variety of languages and libraries. That is the case of TensorFlow, Theano, Keras and Spark, which will have a huge relevance to this document, for the purpose of it is to deal with these resources in order to analyze their performance when they are applied on a type of neural network, the autoencoder, whose basis is to learn the representation (the main features) for an input with a view to regenerate that original data from the acquired characteristics.

**Keywords:** Autoencoders; neural networks; machine learning; deep learning; feature extraction; dimensionality reduction; learning; performance analysis; TensorFlow; Theano; Keras; Spark; Elephas.

## C.2. State of art

### C.2.1. Introduction to neural networks

From a historical perspective, neural networks were firstly brought up in the 1940s by Warren McCulloch and Walter Pitts, who introduced a concept based on neurons, the perceptron, that took some binary inputs and then summed them, returning 1 or 0 depending on whether or not the sum surpassed certain threshold previously defined [6]. Over the years, critics appeared regarding their alleged utility [29], however, new projects were developed based on that initial work, which they improved, shaping the current idea of neural networks by adding new features, such as activation function, algorithms for the initialization of different variables [45] ...

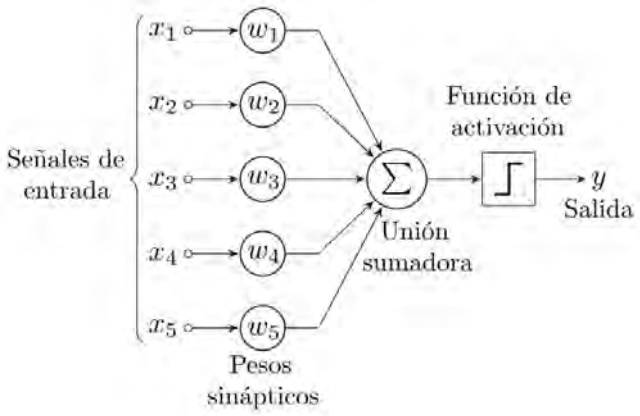



Fig. C.1. Diagram of a perceptron, the simplest kind of network, introduced in the 1940s [23].

The current definition for neural or deep networks is that it is a group of algorithms, trained for pattern recognition in data, which then they process by comparing it to other information like labels [55]. That training can be differentiated into two main methods:

- Supervised training: it consists of learning through a training dataset expressed in such a way that a label or category corresponds to certain features. Later on a test dataset is passed through the network in order to classify it [26].
- Unsupervised training: in this case there is not a training dataset, so the network will try to extract features from the information that describe the way in which it is structured [26].

### C.2.2. Autoencoders

The autoencoder is a kind of neural network based on learning a compressed representation of the data with which it is being fed, in order to try to regenerate from the condensed characteristics that input as much as possible [63].

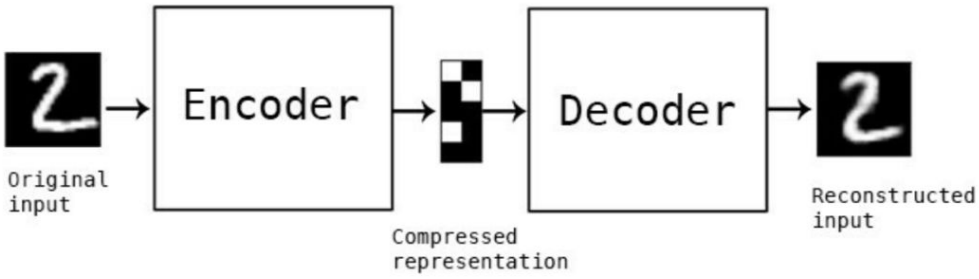



Fig. C.2. Structure of an autoencoder [63].

Generally, the hidden layers, the ones between the input and the output, are smaller than them, thus obtaining a compressed representation of the entered data. In addition, these are some relevant variations of the autoencoder:

- Variational autoencoder: in this case the network learns the statistical properties, mean and variance, that represent the data [64]. This way it can generate random variations of that information, such as people's faces [66] or little musical excerpts [67].
- Denoising autoencoder: this type of network works by adding noise to the inputs and then being trained to remove it, retrieving the original data [68].

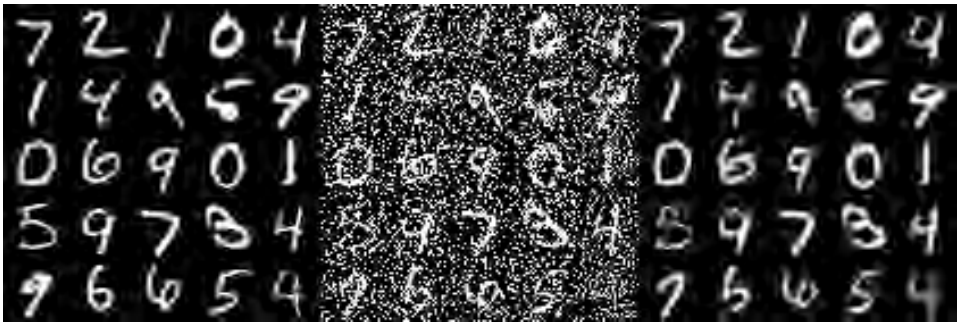


Fig. C.3. The functioning of a denoising autoencoder. Left: original MNIST dataset. Center: noisy images. Right: reconstructed data [70].

### C.2.3. Tools for neural network programming

#### Programming languages

- Python: is one of the most commonly used languages, not only for the programming of neural networks [9]. This is due to the fact that expressing commands and algorithms is relatively simple, which makes its reading and understanding easier. In addition, it contains a large amount of libraries and extensions that allow the user to accomplish tasks in a simpler way.
- R: is another popular language, used mostly for its statistical approach, which makes it possible to carry out deep data analysis in a straightforward manner, and to combine them with a whole variety of visualising options [72].

#### Libraries and frameworks

- TensorFlow: this library, which was developed by Google, is one of most used ones, and it is mainly employed to carry out numeric calculations through data flux diagrams, as well as deep learning-related problems [10].
- Theano: is older than TensorFlow, making it profusely documented and thus, used [11]. However, its creators announced back in 2017 that there would not be any new features added, so its usage is expected to decrease.
- Keras: this library can work with TensorFlow and Theano as its backend [12]. Its intuitive interface has made it increasingly popular in recent years, specially for neural network programming.
- Apache Spark: it is a framework more focused on Big Data-related questions, and offers a fast and user-friendly interface to interact with Python, Java and other languages [14].
- Elephas: this is an extension of Keras that allows it to handle deep learning distributed models with Spark [13].

### C.2.4. Recent work

Neural networks are really common in daily life, as they are already present in the search and recommendation engines of video on demand applications, such as Netflix [78]

or HBO, e-commerce websites like Amazon, and social networks, for instance, Facebook or Twitter.

In addition, deep learning can be applied to the vast majority of sectors, such as education, healthcare, economy, transport, communications and military; as its usefulness and applications are transversal to all of them. Those uses include face, object and voice recognition; prediction of an outcome given some facts, analysis of a large quantity of information and automation of processes.

## C.3. Design and implementation of the solution

### C.3.1. Common features of the solutions

The experiments that were carried out have some shared characteristics, mainly the structure of the network, which will be represented in the next figure, and that states the size of each layer:

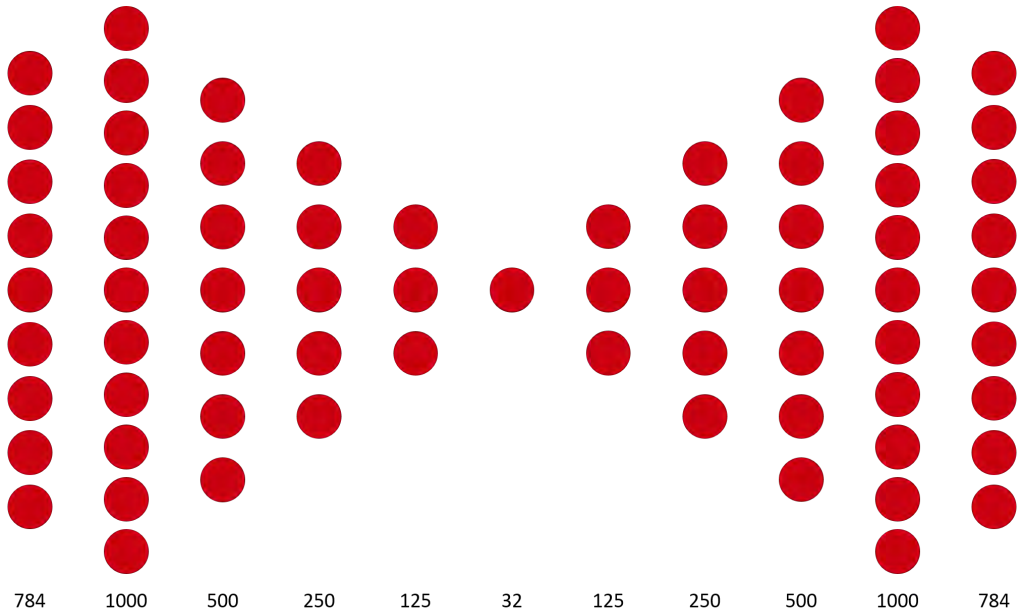



Fig. C.4. Proposed structure of the autoencoder.

Additionally, the tests carried were run on core farms, using 4 CPU, plus 1 GPU in the ones that followed a linear execution.

## C.4. Experiments performed

The main experiments were run with the features presented at the next table:

| Experiment | Batch size | Epochs | Weight initialization | Bias initialization | Learning rate | Activation function |
|---|---|---|---|---|---|---|
| I | 240 | 1000 | Random uniform | Random uniform | 0.1 | Sigmoid |
| II | 240 | 1000 | Random uniform | Random uniform | 0.01 | Sigmoid |
| III | 240 | 1000 | Random uniform | Random uniform | 0.001 | Sigmoid |
| IV | 240 | 1000 | Random normal | Random normal | 0.001 | Sigmoid |
| V | 500 | 1000 | Random normal | Random normal | 0.001 | Sigmoid |
| VI | 500 | 1000 | Random normal | Random normal | 0.001 | ReLU + Sigmoid |
| VII | 500 | 1000 | Xavier normal | Xavier normal | 0.001 | ReLU + Sigmoid |
| VIII | 500 | 1000 | Xavier normal | Xavier normal | 0.0001 | ReLU + Sigmoid |

Tabla C.1. Detailed summary of the main experiments performed.

Regarding the tests carried out, each of them tries to outperform the previous one by varying the parameters aforementioned. As might have been expected, not all of those changes turned out successful, so those experiments became part of the secondary ones, which will not be available here, but in the main paper.

In order to represent all the other tests, the seventh, which achieved the best results, is presented hereafter.

| | |
|---|---|
| Learning rate | 0.001 |
| Batch size | 500 |
| Number of epochs | 1000 |
| Initialization of weights | Xavier normal |
| Initialization of bias | Xavier normal |
| Activation function | ReLU in every layer, except sigmoid in the latter |
| Optimizer | Adam |

Tabla C.2. Main features for the experiment VII

By applying those characteristics on the autoencoder that was presented earlier, and training it with MNIST and Fashion-MNIST databases, the following figures have been obtained.

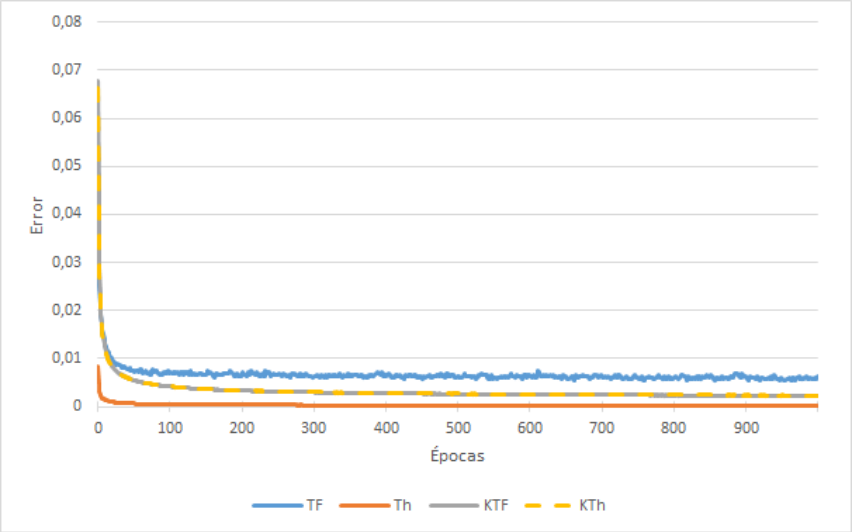



Fig. C.5. Results of experiment VII using MNIST database

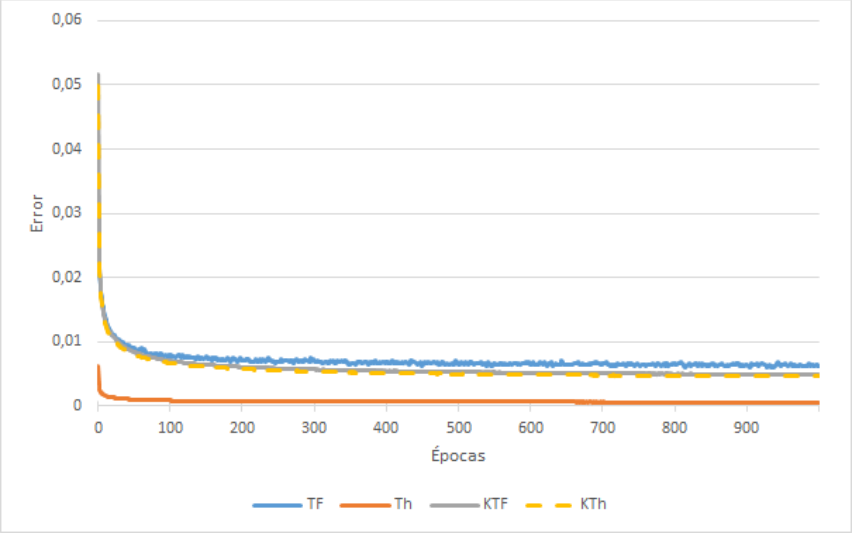



Fig. C.6. Results of experiment VII using Fashion-MNIST database

As can be seen from the pictures above, Theano offers in both cases clearly the best performance. Regarding TensorFlow's results, the oscillating behavior shown mainly with MNIST indicate that it might be necessary to lower the learning rate in the next experiment. In addition, in this test it gave the worst outcome, though with Fashion-MNIST it came close to match Keras'. Finally, the performance Keras provided did not change with the different backends available, and its results, while closer to TensorFlow's, were better.

## C.5. Analysis of results and conclusion

Firstly, a table containing the performance the main experiments offered regarding execution time is presented:

| Experiment | MNIST | | | | FMNIST | | | |
|---|---|---|---|---|---|---|---|---|
| | TF | Th | Keras TF | Keras Th | TF | Th | Keras TF | Keras Th |
| I | 776.62 | 655.89 | 1889.49 | 738.22 | 776.23 | 638.79 | 1856.86 | 755.05 |
| II | 789.17 | 659.39 | 1829.65 | 726.02 | 792.08 | 648.96 | 1839.69 | 713.89 |
| III | 782.60 | 671.98 | 1808.46 | 730.50 | 777.41 | 645.76 | 1870.41 | 723.72 |
| IV | 773.39 | 658.88 | 1913.38 | 713.65 | 775.67 | 651.87 | 1963.81 | 737.19 |
| V | 483.38 | 480.29 | 1087.51 | 505.37 | 485.85 | 498.97 | 1069.71 | 533.00 |
| VI | 507.90 | 538.69 | 1097.73 | 575.12 | 499.38 | 542.71 | 1125.20 | 573.56 |
| VII | 499.59 | 538.37 | 1118.95 | 590.60 | 502.32 | 541.11 | 1140.26 | 590.29 |
| VIII | 770.91 | 554.82 | 1111.01 | 571.09 | 774.18 | 549.65 | 1082.90 | 600.26 |

Tabla C.3. User times of execution for each test, expressed in seconds.

Regarding the libraries used, from the comparison between TensorFlow and Theano, it was the latter which clearly gave the best outcome in terms of results, because this difference was not so clear looking at the time, as both were similar overall. With that in mind, it follows that Theano turns out to be the best option between the two.

The addition of Keras does not add anything significant to the native implementations of TensorFlow and Theano in the light of the outcome. It is worth noting that the performance, while not different regarding the results, is very differential in terms of time, as Keras with Theano spends nearly half the time in the execution than Keras with TensorFlow. However, as it was said early on, Theano offers the best outcome of the four implementations of the autoencoder.

Concerning the sole use of CPU and the distributed execution, the user times obtained are the following:

| Experiment | MNIST | | | | FMNIST | | | |
|---|---|---|---|---|---|---|---|---|
| | TF | Th | Keras TF | Keras Th | TF | Th | Keras TF | Keras Th |
| GPU+CPU | 499.59 | 538.37 | 1,118.95 | 590.60 | 502.32 | 541.11 | 1,140.26 | 590.29 |
| CPU | 57,926.20 | 352,571.79 | 68,591.34 | 267,354.78 | 54,791.47 | 337,546.89 | 65,952.47 | 268,472.80 |

Tabla C.4. User times using CPU plus GPU against CPU for the experiment VII, expressed in seconds.

As can be appreciated from the above table, restricting the use of cores to CPU affects substantially Theano in a negative way, regardless of whether it is used natively or with Keras. As to the results, they do not vary from the ones obtained with the joint use of CPU plus GPU.

Regarding Elephas, it is worth noting as a plus point for Keras the fact that it admits extensions such as the aforementioned, that allow it to run distributed models with Spark. This addition improves the time consumed as it was expected, but it worsens lightly the results up to the point where it starts to raise. To conclude, whether or not to use Elephas depends on which part will be allowed to get worse in order to keep the best results in the other one.

### C.5.1. Future work

Some tasks that stayed out of the paper for different reasons are trying to use GPU with Elephas and carrying out a deeper analysis, including other network structures, different parameters, such as optimizers, initialization algorithms or activation functions; new databases and even concrete applications of autoencoders like variational or denoising.

# BIBLIOGRAFÍA


[1] *Rise of the Machines: 10 Defining Moments in the History of AI*. URL: https://www.digitaltrends.com/cool-tech/history-of-ai-milestones/. Accedido el 05/01/19.

[2] *IoT for Smart Home and City*. URL: https://www.st.com/en/applications/iot-for-smart-home-and-city.html. Accedido el 02/01/19.

[3] *Big Data: qué es y por qué es importante*. URL: https://www.sas.com/es_es/insights/big-data/what-is-big-data.html. Accedido el 03/01/19.

[4] *6 Trends to Get Excited About for the Future of IoT*. URL: https://skelia.com/articles/6-trends-get-excited-future-iot/. Accedido el 02/01/19.

[5] *Gartner: Top 10 Strategic Technology Trends for 2019*. URL: https://datavizblog.com/2018/11/18/gartner-top-10-strategic-technology-trends-for-2019/. Accedido el 05/01/19.

[6] W. McCulloch y W. Pitts. "A logical calculus of the ideas immanent in nervous activity". En: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115-133.

[7] J. Schmidhuber. "Deep learning in neural networks: An overview". En: *Neural networks* 61 (2015), pp. 85-117.

[8] K. Oh y K. Jung. "GPU implementation of neural networks". En: *Pattern Recognition* 37.6 (2004), pp. 1311-1314.

[9] *Página principal de Python*. URL: https://www.python.org/. Accedido el 05/01/19.

[10] *Página principal de TensorFlow*. URL: https://www.tensorflow.org/. Accedido el 05/01/19.

[11] *Página principal de Theano*. URL: http://deeplearning.net/software/theano/. Accedido el 05/01/19.

[12] *Página principal de Keras*. URL: https://keras.io/. Accedido el 05/01/19.

[13] *Página de Elephas en GitHub*. URL: https://github.com/maxpumperla/elephas. Accedido el 05/01/19.

[14] *Página principal de Apache Spark*. URL: https://spark.apache.org/. Accedido el 05/01/19.

[15] *Página principal de la librería PySpark*. URL: https://spark.apache.org/docs/0.9.0/python-programming-guide.html. Accedido el 05/01/19.

[16] *Página principal del dataset MNIST*. URL: http://yann.lecun.com/exdb/mnist/. Accedido el 07/01/19.

[17] *Página principal del dataset FMNIST*. URL: https://github.com/zalandoresearch/fashion-mnist. Accedido el 07/01/19.

[18] *MNIST*. URL: https://en.wikipedia.org/wiki/MNIST_database. Accedido el 07/01/19.

[19] *IMRyD*. URL: https://en.wikipedia.org/wiki/IMRAD. Accedido el 07/01/19.

[20] *F. Rosenblatt*. URL: https://en.wikipedia.org/wiki/Frank_Rosenblatt. Accedido el 10/01/19.

[21] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain". En: *Psychological Review* 65 (1958), pp. 386-408.

[22] D. Hebb. *The organization of behavior. A neuropsychological theory*. John Wiley, 1949.

[23] *Perceptrón*. URL: https://es.wikipedia.org/wiki/Perceptr%C3%B3n. Accedido el 11/01/19.

[24] *A neural networks deep dive*. URL: https://developer.ibm.com/articles/cc-cognitive-neural-networks-deep-dive/. Accedido el 11/01/19.

[25] *A "Brief" History of Neural Nets and Deep Learning*. URL: http://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-and-deep-learning/. Accedido el 11/01/19.

[26] A. Pérez. *Practical Artificial Intelligence: Machine Learning, Bots, and Agent Solutions Using C#*. Apress, 2018.

[27] B. Widrow. *An Adaptive "ADALINE" Neuron Using Chemical "Memistors"*. 1960.

[28] *Machine Learning FAQ: What is the difference between a Perceptron, Adaline, and neural network model?* URL: https://sebastianraschka.com/faq/docs/diff-perceptron-adaline-neuralnet.html. Accedido el 13/01/19.

[29] M. Minsky y S. Papert. "Perceptrons: An Introduction to Computational Geometry". En: (1969).

[30] *Invierno de la IA*. URL: https://en.wikipedia.org/wiki/AI_winter. Accedido el 13/01/19.

[31] P. Werbos. "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences". En: (1974).

[32] D. Rumelhart, G. Hinton y R. Williams. "Learning representations by back-propagating errors". En: *nature* 323.6088 (1986), p. 533.

[33] *Yann LeCun's Research and Contributions*. URL: http://yann.lecun.com/ex/research/index.html. Accedido el 15/01/19.

[34] Y. LeCun et al. "Backpropagation applied to handwritten zip code recognition". En: *Neural computation* 1.4 (1989), pp. 541-551.

[35] D. Rumelhart, G. Hinton y R. Williams. *Learning internal representations by error propagation*. Inf. téc. Institute for Cognitive Science of University of California, 1985.

[36] *From Autoencoder to Beta-VAE*. URL: https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html. Accedido el 15/01/19.

[37] *A "Brief" History of Neural Nets and Deep Learning. Part 2*. URL: http://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-and-deep-learning-part-2/. Accedido el 15/01/19.

[38] D. Ackley, G. Hinton y T. Sejnowski. "A learning algorithm for Boltzmann machines". En: *Cognitive science* 9.1 (1985), pp. 147-169.

[39] R. Neal. "Connectionist learning of belief networks". En: *Artificial intelligence* 56.1 (1992), pp. 71-113.

[40] D. Pomerleau. “Alvinn: An autonomous land vehicle in a neural network”. En: *Advances in neural information processing systems*. 1989, pp. 305-313.

[41] G. Tesauro. “Temporal difference learning and TD-Gammon”. En: *Communications of the ACM* 38.3 (1995), pp. 58-68.

[42] *A “Brief” History of Neural Nets and Deep Learning. Part 3*. URL: http://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-and-deep-learning-part-3/. Accedido el 15/01/19.

[43] T. Robinson, M. Hochberg y S. Renals. “The Use of Recurrent Neural Networks in Continuous Speech Recognition”. En: (ene. de 1995).

[44] *A “Brief” History of Neural Nets and Deep Learning. Part 4*. URL: http://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-and-deep-learning-part-4/. Accedido el 16/01/19.

[45] G. Hinton, S. Osindero e Y. Teh. “A fast learning algorithm for deep belief nets”. En: *Neural computation* 18.7 (2006), pp. 1527-1554.

[46] Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[47] Y. Bengio, Y. LeCun et al. “Scaling learning algorithms towards AI”. En: *Large-scale kernel machines* 34.5 (2007), pp. 1-41.

[48] A. Mohamed, G. Dahl y G. Hinton. “Deep Belief Networks for phone recognition”. En: *Nips workshop on deep learning for speech recognition and related applications*. Vol. 1. 9. Vancouver, Canada. 2009, p. 39.

[49] *Leading breakthroughs in speech recognition software at Microsoft, Google, IBM*. URL: https://www.utoronto.ca/news/leading-breakthroughs-speech-recognition-software-microsoft-google-ibm. Accedido el 16/01/19.

[50] R. Raina, A. Madhavan y A. Ng. “Large-scale deep unsupervised learning using graphics processors”. En: *Proceedings of the 26th annual international conference on machine learning*. ACM. 2009, pp. 873-880.

[51] G. Hinton et al. “Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups”. En: *IEEE Signal processing magazine* 29.6 (2012), pp. 82-97.

[52] *Google Needs To Make Machine Learning Their Growth Fuel*. URL: https://www.forbes.com/sites/louiscolumbus/2018/08/26/google-needs-to-make-machine-learning-their-growth-fuel/#249619f2431b. Accedido el 16/01/19.

[53] *The mostly complete chart of Neural Networks, explained*. URL: https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464. Accedido el 19/01/19.

[54] *How Alexa Is Learning to Converse More Naturally*. URL: https://developer.amazon.com/es/blogs/alexa/post/15bf7d2a-5e5c-4d43-90ae-c2596c9cc3a6/how-alexa-is-learning-to-converse-more-naturally. Accedido el 19/01/19.

[55] *A Beginner's Guide to Neural Networks and Deep Learning*. URL: https://skymind.ai/wiki/neural-network%5C#concept. Accedido el 19/02/19.

[56] *Deep Learning: Overfitting*. URL: https://towardsdatascience.com/deep-learning-overfitting-846bf5b35e24. Accedido el 19/02/19.

[57] *Memorizing is not learning - 6 tricks to prevent overfitting in machine learning*. URL: https://hackernoon.com/memorizing-is-not-learning-6-tricks-to-prevent-overfitting-in-machine-learning-820b091dc42. Accedido el 19/02/19.

[58] *Supervised and Unsupervised Machine Learning Algorithms*. URL: https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/. Accedido el 19/01/19.

[59] *Titanic: Machine Learning from Disaster*. URL: https://www.kaggle.com/c/titanic/data. Accedido el 17/01/19.

[60] *STL-10 Image Recognition Dataset*. URL: https://www.kaggle.com/jessicali9530/stl10. Accedido el 18/01/19.

[61] *Difference Between Supervised, Unsupervised, & Reinforcement Learning*. URL: https://blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/. Accedido el 19/01/19.

[62] I. da Silva, D. Spatti, R. Flauzino, L. Liboni y S. dos Reis Alves. "Artificial neural network architectures and training processes". En: *Artificial Neural Networks*. Springer, 2017, pp. 21-28.

[63] *Building Autoencoders in Keras*. URL: https://blog.keras.io/building-autoencoders-in-keras.html. Accedido el 20/01/19.

[64] *Intuitively Understanding Variational Autoencoders*. URL: https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf. Accedido el 20/01/19.

[65] *Variational Autoencoders Explained*. URL: http://kvfrans.com/variational-autoencoders-explained/. Accedido el 20/01/19.

[66] *Variational auto-encoder trained on celebA*. URL: https://github.com/yzwxx/vae-celebA. Accedido el 20/01/19.

[67] A. Roberts, J. Engel y D. Eck. "Hierarchical variational autoencoders for music". En: *NIPS Workshop on Machine Learning for Creativity and Design*. 2017.

[68] A. Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2017.

[69] *Introduction to autoencoders*. URL: https://www.jeremyjordan.me/autoencoders/. Accedido el 21/01/19.

[70] *Tutorial: Your First Model (DAE)*. URL: http://www.opendeep.org/v0.0.5/docs/tutorial-your-first-model. Accedido el 21/01/19.

[71] A. Makhzani y B. Frey. "k-Sparse Autoencoders". En: *arXiv preprint arXiv:1312.5663* (2013).

[72] *The R Project for Statistical Computing*. URL: https://www.r-project.org/. Accedido el 10/05/19.

[73] *MATLAB - MathWorks*. URL: https://www.mathworks.com/products/matlab.html. Accedido el 10/05/19.

[74] *SQL*. URL: https://en.wikipedia.org/wiki/SQL. Accedido el 10/05/19.

[75] *MILA and the future of Theano*. URL: https://groups.google.com/forum/%5C#!msg/theano-users/7Poq8BZutbY/rNCIfvAEAwAJ. Accedido el 22/01/19.

[76] *Comparison of AI Frameworks*. URL: https://skymind.ai/wiki/comparison-frameworks-dl4j-tensorflow-pytorch. Accedido el 22/01/19.

[77] *Spark vs Hadoop, ¿quién saldrá vencedor?* URL: https://blog.powerdata.es/el-valor-de-la-gestion-de-datos/spark-vs-hadoop-quien-saldra-vencedor. Accedido el 22/01/19.

[78] C. Gomez-Uribe y N. Hunt. “The Netflix Recommender System: Algorithms, Business Value, and Innovation”. En: *ACM Trans. Manage. Inf. Syst.* 6.4 (dic. de 2015), 13:1-13:19.

[79] *Facial recognition software is highly effective when paired with social media platform*. URL: http://rijock.blogspot.com/2017/10/facial-recognition-software-is-highly.html. Accedido el 22/01/19.

[80] N. Jouppi, C. Young, N. Patil y D. Patterson. “A domain-specific architecture for deep neural networks”. En: *Communications of the ACM* 61.9 (2018), pp. 50-59.

[81] *Ley de Amdahl*. URL: https://es.wikipedia.org/wiki/Ley_de_Amdahl. Accedido el 20/02/19.

[82] L. Moreno et al. “Tema 3. Leyes sobre el aumento de prestaciones”. En: *Organización de Computadores* ().

[83] S. Yu y J. Príncipe. “Understanding autoencoders with information theoretic concepts”. En: *Neural Networks* 117 (2019), pp. 104-123. URL: http://www.sciencedirect.com/science/article/pii/S0893608019301352.

[84] *RDD Programming Guide*. URL: https://spark.apache.org/docs/latest/rdd-programming-guide.html. Accedido el 08/06/19.

[85] *Apache Mesos*. URL: http://mesos.apache.org/. Accedido el 08/06/19.

[86] S. Smith, P. Kindermans, C. Ying y Q. Le. “Don’t decay the learning rate, increase the batch size”. En: *arXiv preprint arXiv:1711.00489* (2017).

[87] *Neural Networks: A Modern Introduction*. URL: https://compsci682.github.io/notes/neural-networks-1/. Accedido el 04/06/19.

[88] X. Glorot e Y. Bengio. “Understanding the difficulty of training deep feedforward neural networks”. En: (2010), pp. 249-256.

[89] K. He, X. Zhang, S. Ren y J. Sun. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”. En: *2015 IEEE International Conference on Computer Vision (ICCV)* (2015). Accedido el 07/05/19.

[90] *Types of Optimization Algorithms used in Neural Networks and Ways to Optimize Gradient Descent*. URL: https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f. Accedido el 07/06/19.

[91] *ML | Stochastic Gradient Descent (SGD)*. URL: https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/. Accedido el 29/05/19.

[92] *An Introduction to AdaGrad*. URL: https://medium.com/konvergen/an-introduction-to-adagrad-f130ae871827. Accedido el 02/06/19.

[93] M. Zeiler. "ADADELTA: an adaptive learning rate method". En: *arXiv preprint arXiv:1212.5701* (2012).

[94] *Understanding RMSprop: faster neural network learning*. URL: https://towardsdatascience.com/understanding-rmsprop-faster-neural-network-learning-62e116fcf29a. Accedido el 03/06/19.

[95] *Norma del supremo*. URL: https://es.wikipedia.org/wiki/Norma_del_supremo. Accedido el 03/06/19.

[96] D. Kingma y J. Ba. "Adam: a method for stochastic optimization". En: *3rd International Conference for Learning Representations* (2015).

[97] *A Practical Guide to ReLU*. URL: https://medium.com/tinymind/a-practical-guide-to-relu-b83ca804f1f7. Accedido el 29/04/19.

[98] *Activation Functions in Neural Networks*. URL: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. Accedido el 04/05/19.

[99] *Reglamento General de Protección de Datos*. Reglamento (UE) 2016/679 del Parlamento Europeo y del Consejo, de 27 de abril de 2016 https://eur-lex.europa.eu/legal-content/ES/TXT/?uri=CELEX:32016R0679.

[100] *Ley Orgánica de Protección de Datos de Carácter Personal*. BOE núm. 298, de 14 de diciembre de 1999 https://www.boe.es/buscar/doc.php?id=BOE-A-1999-23750.

[101] *Ley Orgánica de Protección de Datos Personales y garantía de los derechos digitales*. BOE núm. 294, de 6 de diciembre de 2018 https://www.boe.es/eli/es/lo/2018/12/05/3/con.

[102] P. Mozur. *Inside China's Dystopian Dreams: A.I., Shame and Lots of Cameras*. URL: https://www.nytimes.com/2018/07/08/business/china-surveillance-technology.html?module=inline. Accedido el 09/06/19.

[103] P. Mozur, J. Kessel y M. Chan. *Made in China, Exported to the World: The Surveillance State*. URL: https://www.nytimes.com/2019/04/24/technology/ecuador-surveillance-cameras-police-government.html. Accedido el 09/06/19.

[104] L. Gatys, A. Ecker y M. Bethge. "Image style transfer using convolutional neural networks". En: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2414-2423.

[105] *DeepDream*. URL: https://en.wikipedia.org/wiki/DeepDream. Accedido el 09/06/19.

[106] *Draw to Art | Experiments with Google*. URL: https://experiments.withgoogle.com/draw-to-art. Accedido el 09/06/19.

[107] *X Degrees of Separation | Experiments with Google*. URL: https://experiments.withgoogle.com/x-degrees-of-separation. Accedido el 09/06/19.

[108] *Life Tags | Experiments with Google*. URL: https://experiments.withgoogle.com/life-tags. Accedido el 09/06/19.

[109] *Inceptionism: Going deeper into Neural Networks*. URL: https://photos.google.com/share/AF1QipPX0SCl7OzWilt9LnuQliattX4OUCj_8EP65_cTVnBmS1jnYgsGQAieQUc1VQWdgQ?key=aVBxWjhwSzg2RjJWLWRuVFBBZEN1d205bUdEMnhB. Accedido el 09/06/19.

[110] *Instancias P3 de Amazon EC2*. URL: https://aws.amazon.com/es/ec2/instance-types/p3/. Accedido el 10/06/19.